VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Distribuovaný databázový systém založený na MapReduce

Distributed DBMS based on MapReduce

2014 Adam Babušek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Adam Babušek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Distribuovaný databázový systém založený na MapReduce
Distributed DBMS based on MapReduce

Zásady pro vypracování:

MapReduce je programovací model pro zpracování a generování velkého množství dat. Úkolem diplomové práce je identifikovat klíčové vlastnosti a výhody toho modelu oproti ostatním modelům v distribuovaném prostředí.

1. Nastudujte programovací model MapReduce.
2. Navrhněte a naimplementujte distribuovaný databázový systém s využitím tohoto modelu.
3. Pro zvolená data a dotazy proveďte experimenty, výsledky vyhodnoťte a porovnejte s dalšími distribuovanými databázovými systémy.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Michal Krátký, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7.5. 2014

Adam Baleutel

Rád bych velmi poděkoval panu doc. Ing. Michalu Krátkému, Ph.D. za jeho odbornou a trpělivou pomoc při vedení této diplomové práce. Dále poděkování patří všem blízkým za jejich podporu.

Abstrakt

Práce je zaměřena na vývoj distribuovaného databázového systému implementující model MapReduce pro zpracování rozsáhlých datových kolekcí v distribuovaném prostředí. Vysvětluje princip fungování modelu, jeho využití v distribuovaném prostředí a následně popisuje vybrané systémy, které jsou zcela nebo jen částečně založeny na tomto modelu. Na základě teoretických poznatků je dále navrhnut a naimplementován distribuovaný systém založený na tomto modelu. Systém je otestován a porovnán s jiným vybraným systémem. Podle výsledků testů je vyhodnocena vhodnost návrhu implementace a jsou uvedeny možná rozšíření systému.

Klíčová slova: model MapReduce, distribuované SŘBD, nerelační SŘBD, dotazovací jazyk SQL

Abstract

Thesis is focused on the development of distributed database system implementing the MapReduce model for processing large data collections in distributed environment. It explains principle of operation of the model, its functionality in a distributed environment and describes selected systems which are fully or partially based on this model. Based on theoretical knowledge is designed and implemented distributed system using the MapReduce model. System is tested and compared with other selected system. According to test results is evaluated the suitability of the design and implementation and described the possible expansion of the system.

Keywords: MapReduce model, distributed DBMS, non-relational DBMS, query language SQL

Seznam použitých zkratek a symbolů

OSDI	–	Operating Systems Design & Implementation
SAN	–	Storage Area Network
SN	–	Shared-Nothing
SQL	–	Structured Query Language
RDBMS	–	Relational Database Management System
HPC	–	High Performance Computin
MPI	–	Message Passing Interface
HDFS	–	Hadoop Distributed Filesystem
JSON	–	JavaScript Object Notation
BSON	–	Binary JavaScript Object Notation
REST	–	Representational State Transfer
MVCC	–	Multi-Version Concurrency Control
OLTP	–	OnLine Transaction Processing
OLAP	–	OnLine Analytical Processing
CLI	–	Command Language Interface
GFS	–	Google File System
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
VPN	–	Virtual Private Network

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Řešení problematiky uchování, zpracování či prezentace rozsáhlých kolekcí dat je dnes každodenní práce mnoha společností. V průběhu let byly vyvinuty různé systémy řízení báze dat sloužící ať už v komerčním nebo vědeckém prostředí. S rostoucím objemem dat se systémy začaly do určité míry decentralizovat, aby vyhověly náročným požadavkům na dostupnost, bezpečnost, škálovatelnost a rychlost přenosu či zpracování dat. Jen známé internetové společnosti jako jsou např. Amazon, Google, Facebook, Yahoo, atd., uchovávají na svých záznamových médiích stovky terabajtů nebo i jednotky či desítky pentabajtů dat. Jen zaznamenávání aktivity stovkami milionů uživatelů může generovat stovky terabajtů dat denně. Takto masivní objem dat se obecně nazývá *Big Data*. Jejich analýza přináší mnohé poznatky využitelné nejen v komerční sféře, ale i vědě. Příkladem může být analýza sociální sítě, kdy data obsahují grafové údaje umožňující zkoumání interakcí a chování jednotlivců na základě konkrétních podnětů. Metody samotné analýzy dat řeší disciplína zvaná *Data Mining*. Aby analýza proběhla v rozumném čase, začala být s nárůstem objemu a distribuovanosti dat aktuální problematika dotazování a zpracování Big Data. MapReduce [1] je jedním z řešešní problematiky využitelný právě v decentralizovaném prostředí.

Programovací model MapReduce je svým původem určen pro nasazení především v síti skládající se z autonomních uzlů, které obsahují část dat, případně jejich kopie. Jeho primárním úkolem je zajistit jednodušší a spolehlivější zpracování v takovém prostředí.

Práce je rozdělena na pět významných částí. V druhé kapitole je obecně popsán model MapReduce, podrobněji vysvětleno, jak probíhá zpracování konkrétního dotazu nad daty a následně identifikujeme klíčové vlastnosti modelu a porovnáme s jinými známými modely. Třetí kapitola stručně popisuje specifika vybraných volně dostupných systémů, které obsahují implementaci tohoto modelu. Mírně jsou také zmíněny komerčních systémy. Získáme tak stručný přehled o možnostech a funkčnosti dále popsaných systémů a dle zájmu i směr, kterým se čtenář může vydat při dalším studiu probírané problematiky. Na základě získaných teoretických poznatků je čtvrté kapitole popsána analýza, návrh a implementace vlastního systému s využitím popisovaného modelu. Návrh a implementace takového systému nám pomůže lépe identifikovat zásadní problémy, které je nutné při implementaci řešit. Pátá kapitola obsahuje popis, výsledky a vyhodnocení prováděných testů systému nad vybranými daty. Testy jsou ověřením správnosti návrhu systému, který je i porovnán s jiným vybraným systémem. Poslední kapitola pak obsahuje shrnutí práce a možná doporučení rozšíření nebo změn v návrhu vyplývající z provedených testů.

Základní referencí pro studium modelu je jako zdroj využita zejména literatura *MapReduce: Simplified Data Processing on Large Clusters* [1], *Hadoop: The Definitive Guide* [3] a *A Comparison of Approaches to Large-Scale Data Analysis* [2], která může dobře posloužit jako vstupní literatura případným dalším čtenářům této práce se zájmem o problematiku MapReduce.

2 Model MapReduce

Jak bylo v úvodu zmíněno, rozsáhlé datové kolekce jsou často označovány slovním spojením Big Data [32]. Není specifikováno přesně od jaké velikosti či distribuovanosti dat jsou data označovány jako Big Data, ale obecně lze říci, že takto můžeme označit každé datové kolekce, na jejichž zpracování nelze z důvodu jejich velikostí aplikovat klasický přístup analýzy či zpracování pomocí databází na jednom počítači. Jedním z přístupů, jak nejen takto rozsáhlá data zpracovávat, je právě model MapReduce.

2.1 Obecný popis

MapReduce je programovací model představený společností Google roku 2004 na konferenci OSDI [1]. Autoři vyvinuli tento model pro zvětšení míry abstrakce při návrhu aplikací využívající distribuované zpracování rozsáhlých dat. Nechali se inspirovat funkcionálními jazyky (Lisp [23]), ve kterých se využívá dvou funkcí - *mapovací funkce* a *funkce redukce*. Nyní se však map-reduce zpracování využívá i ve vyskoúrovňových jazycích v podobě lamba syntaxe, např. Java 8 [29]. Uživatelem tohoto modelu je programátor, který tyto dvě funkce v rámci modelu implementuje. Myšlenka spočívá v aplikování mapovací funkce na vstupní množinu dat tak, aby byly vytvořeny mezivýsledky typu klíč:hodnota, a dále ve zpracování redukční funkcí, která seskupí hodnoty se stejným klíčem do výsledné množiny.

Dle autorů, kteří představili zpracování dát za pomocí využití MapReduce modelu má tento model využití v prostředí zvaném *shared-nothing architecture* (dále jen SN). Jedná se o distribuované prostředí skládající se z vysokého počtu uzlů běžně dostupného hardwaru (stovky až tisíce), které pracují samostatně a nezávisle na sobě. Každý uzel v clusteru má svůj lokální disk s daty databáze, paměť a výpočetní jednotku. Výhodou modelu je vysoká škálovatelnost - síť je možno neustále rozšiřovat o nové a nové uzly. S tím jsou spojené také nižší náklady na pořízení a provoz, díky používání běžného hardwaru s dvou či více jádrovými procesory narozdíl od nákladných multi-procesorových serverů a sítí *storage area network* (dále jen SAN). SN využívá mnoho známých společností, např. Wikipedia, Facebook, Amazon a spoustu dalších. Implementace MapReduce se však nesoustředí jen na aplikaci modelu v clusteru uzlů, ale lze jej také aplikovat na jednom uzlu s více-jádrovým procesorem či ve více-procesorových systémech [4]. Toto řešení však vyžaduje určitou modifikaci zpracování, a proto se v rámci této práce budeme bavit o aplikaci MapReduce v clusteru nezávislých uzlů.

Vzhledem k vysoké škálovatelnosti v systému MapReduce model předem nevytváří podrobný plán provedení dotazu přidělující konkrétním uzlům konkrétní úlohu. Místo toho jsou úlohy přidělovány postupně až za běhu programu. Program je pak schopen se lépe vypořádat s problémy způsobenými nefunkčními či pomalými uzly. Další výhoda je tedy spolehlivost. Té model dosahuje pomocí detekce nezdařených úloh na nefunkčních uzlech a přerozdělováním těchto úloh na jiné uzly, nejlépe na ty, které obsahují repliky vstupní množiny dat pro mapovací funkci - odpadá tak časová ztráta způsobená dalším přenosem dat. Model je dále schopen pracovat v heterogenním prostředí. Heterogenním se má v tomto případě na mysli rozdílný výkon jednotlivých uzlů v clusteru. Úlohy,

jejichž zpracování trvá příliš dlouho, jsou znovu spouštěny na výkonnějších uzlech, které již dokončili své přiřazené úlohy. Výkonnější uzly tedy postupně zpracují více úloh. Výsledky jednotlivých mapovacích funkcí se pak průběžně zapisují na lokální disky, aby se co nejvíc omezilo množství opakování úloh při případném pádu systému.

O paralelní zpracování úloh, distribuci dat, řešení chybových stavů či jiné úlohy související s problematikou distribuovaných systémů se stará samotná implementace modelu [2], programátor pouze definuje mapovací funkci a funkci redukce, a je tak schopen efektivně vyvíjet aplikace využívající distribuované zpracování dat bez znalosti této problematiky.

2.2 Datový model

MapReduce model spadá do oblasti tzv. *NoSQL* [30] databází. Často se lze dočíst o porovnávání NoSQL a SQL databází. SQL je dotazovací jazyk nad strukturovanými daty vytvořený pro dotazování nad relačními databázemi. V takovém porovnání tedy neporovnáváme dotazovací jazyky, ale typy databází z pohledu struktury jejich dat a především z pohledu vlastností *ACID* [21]. Jedná se o vlastnosti (atomičnost, konzistence, izolovanost a perzistence) jejichž dodržení garantuje databázový systém při vykonávání transakcí. NoSQL systémy se vyznačují tím, že nemusí nesplňovat (často i záměrně např. z důvodu výkonu) některé z těchto vlastností. Zároveň v případě NoSQL se tedy jedná se o databáze jejichž data nemusí splňovat předem definované schéma, které vytváří relace v podobě řádků a sloupců jako je tomu u relačních databází. Data mohou být v podstatě v jakémkoliv formátu, může to být nestrukturovaný text, strukturované csv soubory, soubory s páry klíč:hodnota, dokumentové databáze s objekty JSON či BSON [31] nebo další mnohé typy objektů. Vzhledem k myšlence programovacího modelu, který je nezávislý na vstupních datech, je nutné v modelu implementovat nejen mapovací funkci a funkci redukce, ale také parser, díky kterému je pak model schopen předávat mapovací funkci jednotlivé záznamy. Některé implementace modelu však obsahují předem definované parsovací funkce pro známé typy dat, např. csv soubory či zmiňované JSON objekty.

Vzhledem k výše popsaným informacím o datech v NoSQL databázích je nutné dodat, že pokud se v rámci modelu MapReduce bavíme o "záznamech", rozhodně se nemusí jednat o záznamy tak, jak je známe z relačních databází, tedy o řádek tabulky, respektive instanci entity, ale může to být např. jedna věta ze vstupního textu, barva pixelu v případě vstupních dat jako obrázků, apod.

Ať už je v konkrétní implementaci MapReduce použit jakýkoliv typ dat, pak pro všechny platí, že vstupní data jsou rozdělena do jednotlivých bloků. Důvod a použití tohoto rozdělení je popsán v následující podkapitole o zpracovní úlohy.

2.3 Průběh zpracování úloh

Jak přesně model MapReduce pracuje záleží na konkrétní implementaci, myšlenka modelu však zůstává stejná. V této práci budeme vycházet z popisu zpracování podle popisu autorů Dean Jeffrey a Ghemawat Sanjay [1], ze kterého většina implementací vychází

a díky své jednoduchosti a přehlednosti poslouží jako vhodný vzor k pochopení problému. Konkrétní implementace průběhu zpracování je podrobněji popsána v části práce o návrhu a implementaci systému.

Jak bylo zmíněno, uživatel implementuje v modelu mapovací funkci i funkci redukce. Map funkce vytváří z každého záznamu vstupních dat dvojice typu klíč:hodnota. Jsou to mezivýsledky, které se dále použijí v funkci redukce. Klíč není v tomto kontextu unikátní hodnota, ale hodnota, dle které se ve funkci redukce dvojice sjednotí a dále zpracují. Obecněji lze tedy map-reduce zpracování rozdělit na dvě fáze - *mapovací fáze* a *fáze redukce*. Zjednodušeně lze říci, že základní úlohy prováděné v mapovací fázi jsou:

1. **čtení** - iterace záznamy ze vstupních dat
2. **selekce** - výběr záznamů splňující definovanou podmínku
3. **projekce** - výběr či tranformace hodnot iterovaného záznamu
4. **výstup** - vytvoření mezivýsledků - párů klíč:hodnota

a následně úlohy prováděné v reduce fázi jsou

1. **čtení** - iterace záznamy výstupních dat mapovací fáze
2. **spojení** - seskupení záznamů dle klíče
3. **zpracování** - provedení požadovaného zpracování seskupených dat (typicky provedení agregační funkce typu průměr, medián, počet, apod.)
4. **výstup** - vytvoření výstupních dat

Tyto popsané kroky jsou jen základem k porozumění tomu, jaké hlavní úlohy se provádí v jednotlivých fázích zpracování a samy o sobě nestačí ke kompletnímu provedení úlohy. Abychom si je mohli popsat podrobněji, je potřeba porozumět distribuovanému zpracování úlohy, tedy v síti nezávislých uzlů.

2.3.1 Zpracování dotazu

Při spuštění map-reduce úlohy je k dispozici určitý počet uzlů. Tyto uzly lze rozdělit na dva druhy z pohledu jejich úloh, a sice *master* a *worker* (dále jako master uzel a pracovní uzel). Master je "mozkem"celého zpracování úlohy, řídí ji od přijmutí požadavku až po odeslání výsledných dat. Pracovní uzly pak vykonávají samotné distribuované zpracování a lze je dále rozdělit na další dva druhy - *mapovací uzel*, kde je volána mapovací funkce, a *uzel redukce*, kde je volána funkce redukce. Během zpracování se ve většině případů uzel stává mapovacím uzlem i uzlem redukce.

Jak bylo zmíněno, vstupní data jsou rozdělena do M bloků fixní velikosti, typicky 64 MB, 128 MB či 256 MB (záleží na způsobu použití implementace, vstupních datech, dostupném hardwaru, apod.). Tyto bloky jsou rovnoměrně rozloženy po celém clusteru,

avšak není to podmínka. Pro co nejoptimálnější výkon je však vhodné toto rozdělení provést rovnoměrně, aby každý mapovací uzel zpracoval přibližně stejné množství vstupních dat. Každý mapovací uzel rozdělí své mezivýsledky do *R* částí a to tak, že každá dvojice klíč:hodnota z mapovací funkce je přiřazena dle klíče do jedné z těchto částí. Pro správné zpracování je podmínkou provádět rozdělení tak, aby se konkrétní klíč dvojic nacházel právě v jedné z R částí. Naopak, *i*-tá část (kdy $i <= R$) může obsahovat více typů klíču. Jinými slovy, průnik množin(R částí) klíčů je prázdný. Po ukončení práce všech mapovacích uzlů a tedy i mapovací fáze, se mezivýsledky zasílají na uzly redukce, které provádí další zpracování. Každý z uzlů zpracovává *i*-tou část mezivýsledků každého mapovacího uzlu. Zajistí se tím seskupení všech hodnot pro množinu dvojic se stejným klíčem. Z toho vyplývá, že počet uzlů redukce je roven hodnotě *R*, nastavené na počátků zpracování úlohy. Pro mapovací uzly pak platí, že jejich počet je roven počtu bloků, do kterých jsou rozdělena vstupní data. Je nutné ještě poznamenat, že jak hodnota *M*, tak hodnota *R* může převyšovat počet uzlů v clusteru. Jeden uzel tedy může obsloužit více mapovacích uzlů či uzlů redukce.

Jak distribuované zpracování probíhá je shrnuto v následujících bodech. Zároveň jsou jednotlivé body pro přehlednost zobrazeny na obrázku 1.

1. Jeden z uzlů je určen jako master uzel, ostatní jsou pracovní uzly, kterým master postupně přiděluje úlohy. Každý pracovní uzel ukládá výsledky mapovací funkce do *R* bloků. Master uzel má tedy za úkol přidělit *M* mapovacích úloh a *R* úloh redukce.

2. Pracovní uzel, který má přidělenu mapovací úlohu, načítá data z odpovídajícího bloku dat a mapovací funkce vytvoří mezivýsledky typu klíč:hodnota.

3. Dvojice se průběžně ukládají na disk do již zmíněných *R* bloků. Mapovací uzel pak notifikuje master uzel o ukončení své úlohy, který udržuje informace o zpracovaných částí konkrétním mapovacím uzlem.

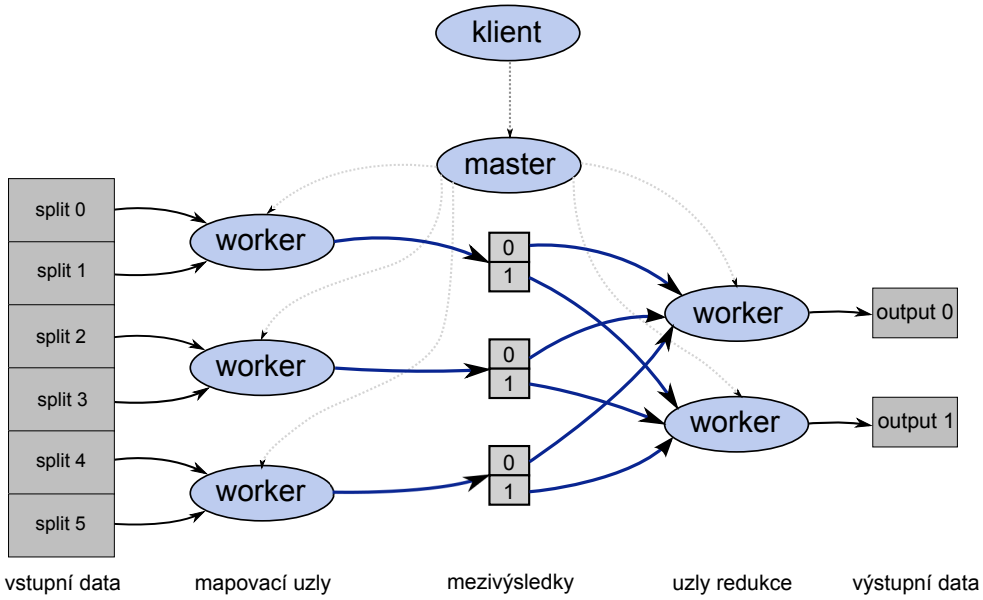
4. Uzel redukce pak obdrží informaci o mapovacích uzlech, jejichž mezivýsledky má zpracovat, respektive část mezivýsledků. Vyžádá si tedy zaslání patřičné části mezivýsledků ze všech mapovacích uzlů, o kterých jej informoval master uzel.

5. Uzel redukce prochází setříděnými daty a funkci redukce předává množinu hodnot se stejným klíčem. Funkce tyto hodnoty zpracuje a výsledky uloží do výstupního souboru.

6. Po dokončení všech pracovních uzlů master uzel ukončí úlohu a dále oznámí tuto skutečnost klientskému programu.

V MapReduce hraje důležitou úlohu replikace, která nám obecně umožňuje větší dostupnost dat a v případě MapReduce dále také

- spolehlivost - při selhání uzlu lze konkrétní úlohu znovu spustit na jiném uzlu,
- vyrovnání zátěže - rychlejší uzly mohou zpracovat více bloků než pomalejší.



Obrázek 1: Zpracování dat v modelu MapReduce

Díky popisu základních úloh jednotlivých uzlů a jejich interakce je v následující podkapitole podrobněji popsána mapovací fáze a fáze redukce probíhající na jednotlivých uzlech.

2.3.2 Mapovací fáze

V map fázi je prvním krokem načítání záznamů z bloku vstupních dat, které jsou předávány mapovací funkci. Jak bylo zmíněno dříve, MapReduce model je nezávislý na typu zdroje dat. Tak jak byl tento model představen, byl jako datové uložiště zvolen distribuovaný souborový system, konkrétně GFS [1]. Tyto souborové systémy umožňují transparentní přístup k souboru na vzdáleném úložišti a navíc obsahují další funkcionality, především automatické rozdělování dat do jednotlivých bloků nebo automatickou replikaci dat. Tato funkcionalita předurčuje distribuované souborové systémy jako vhodné úložiště pro model MapReduce. Není to však podmínkou, neboť v případě nutnosti konkrétní implementace modelu nemusí replikaci využívat a stejně tak i vstupní data nemusí být rozdělena do více bloků, respektive data představují jeden velký blok.

Další uvedené kroky probíhající v map fázi jsou filtrace a selekce. Tyto kroky nejsou uvedeny jako kroky mapovací fáze ve specifikaci modelu, logicky ale vyplývá, že jsou prováděny v mapovací funkci, neboť té jsou předávány jednotlivé záznamy. Výsledky map fáze jsou rozdělovány dle klíče do R bloků. Je podstatné toto rozdělení provádět co nejrovnoměrněji z důvodu zpracování jednotlivých bloků v fázi redukce. Toho samořejmě nelze vždy dosáhnout, záleží na podobě vstupních dat. Jako doporučený způsob určování patřičného bloku zpracovaného výstupu je hodnota zbytku po celočíselném dělení

hashové hodnoty velikostí *R*, tedy

$$block = hash(key) \bmod R$$

Je však důležité použít vhodnou hashovací funkci, která nám umožní zmiňované rovnoměrné rozložení výstupu v R-blocích.

Vzhledem k tomu, že model je vhodný pro zpracováni či analýzu dat, ve většině případech jsou data agregována dle zvolených kritérií. Tuto agregaci provádí funkce redukce, nicméně pokud je agregační funkce komutativní a asociativní, lze ji použít již v mapovací funkci. Komutativní funkce je taková funkce, při které pro dosažení správného výsledku nezáleží na pořadí prvků (např. $sum(a, b) = sum(b, a)$), a asociativní funkce je taková funkce, kterou pro dosažení správného výsledku lze aplikovat na podmnožiny prvků (např. $sum(a, b, c, d) = sum(sum(a, b), sum(c, d))$). Příkladem funkce, která splňuje asociativnost a komutativnost je např. součet nebo počet, naopak funkce nesplňující asociativnost jsou např. průměr nebo medián. Je tedy jasné, v jakých případech lze agregaci použít již v map fázi. Díky použití agregace v mapovací fázi se části mezivýsledků agregují již před zasíláním na uzly redukce a se tím urychlí jak samotný přenos agregovaných mezivýsledků, tak následné zpracování ve fázi redukce.

Podmínkou pro spuštění fáze redukce je ukončení práce všech mapovacích uzlů. Mapovací fáze je tedy tak rychlá, jak rychlý je nejpomalejší pracoví uzel. Tomuto nedostatku lze zčásti zamezit replikací dat, díky které výkonnejší uzel může převzít jednu nebo více úloh méně výkonného uzlu.

2.3.3 Fáze redukce

Jakmile dojde ke spuštění fáze redukce, nastává přesouvání R-částí mezivýsledků z mapovacích uzlů na uzly redukce [3][5]. Každý uzel redukce přijme právě jednu část z každého mapovacícho uzlu. Celkem je tedy uskutečněno $M \times R$ přenosů mezivýsledků. Celkový výkon nejen map-reduce úlohy proto významě ovlivní propustnost sítě v clusteru.

Po přijetí každého *i*-tého bloku (kdy $i <= R$) ze všech mapovacích uzlů proběhne jejich sjednocení. Vzhledem k tomu, že konkrétní typy klíčů se nachází právě v jediném R-tém bloku, data jsou pro daný klíč kompletní a lze provést závěrečnou funkci redukce. Dále vzhledem k přítomnosti více typů klíčů v seskupených blocích je mnohdy nutné data pro správné provedení seskupení dle klíčů setřídit. V případě, že se v mapovací fázi neprovede prvotní agregace dat, může být toto třídení časově náročné a je tedy vhodné data třídit v každém bloku již v mapovací fázi. Sjednocená data v fázi redukce jsou díky tomu předtříděna, což snižuje počet operací některým třídícím algoritmům.

Výsledkem funkce redukce je jeden blok, po skončení celé fáze tedy máme celkem R výsledných bloků dat. Tyto data mohou již sloužit přímo pro prezentaci nebo jako vstupní bloky další map-reduce úlohy.

2.3.4 Join v MapReduce

Relační DBMS provádí spojování tabulek pomocí spojovacích klíčů, kde jsou vytvářeny n-tice kombinující vybrané atributy obou tabulek. Toto řešení se u nerelačních databází

zpravidla neřeší z důvodu neefektivity, obzvlášť pokud se jedná o distribuovanou databázi. Často je toto řešeno pomocí denormalizace, kdy se do dalších tabulek zavádí redundance některých atributů [24]. Mějme např. tabulky *Uživatel* a *Fotografie*. Pokud bychom prováděli nad tabulkami dotaz, jehož výsledkem by byl seznam fotografií všech uživatelů starších pětadvaceti let, při návrhu modelu u nerelační databáze by bylo vhodné přidat atribut (datum_narozeni) nejen do tabulky Uživatel (kde významově patří), ale také do tabulky Fotografie. Tato redundance umožní spustit požadovaný dotaz, nicméně přináší řadu problémů související s redundancí dat. Proto se nerelační systémy zpravidla nepoužívají tam, kde je předpoklad pro časté spojování tabulek.

Model MapReduce však spojování tabulek umožňuje a to dvěma způsoby - *Map-Side Joins* a *Reduce-Side Joins* [3], tedy spojování prováděné v mapovací fázi a spojování provádění ve fázi redukce.

U map-side je nutné implementovat spojení tabulek ještě před dosažení mapovací funkce. Podmínkou je rozdělení tabulek do stejného počtu oddílů, které jsou setříděny podle stejného spojovacího klíče. Postupným procházením tabulek dle klíčů lze dosáhnout vytvoření nového pohledu spojených tabulek, se kterým se dále pracuje obvyklým, výše uvedeným, postupem.

Reduce-side spojování je jednodušší v tom, že spojované tabulky nemusí být stejně rozděleny ani setříděny, nicméně je o něco méně efektivní, protože tabulky se první fázi zpracovávají samostatně a spojovány jsou až ve fázi redukce. Zasíláním mezi uzly tedy prochází větší množství dat. Myšlenka dále spočívá v označením každé dvojice klíč-hodnota jejich zdrojovou tabulkou v map fázi, kdy klíč je spojovací klíč tabulek a hodnota je jeden či více zvolených atributů. Ve fázi redukce před započetím funkce redukce je pak vytvořen nový pohled postupným procházením dvojic podle jejich zdrojové tabulky.

Tento popis spojování tabulek pomocí MapReduce je zde popsán velmi stručně, jeho problematika je mnohem rozsáhlejší [24][25]. Detailním popisem algoritmizace spojování tabulek v modelu MapReduce se tedy dále nebudeme zabývat.

2.4 Nevýhody modelu

Z teorie o modelu MapReduce lze již vyznačit některá slabá místa. Jako první slabé místo se může jevit nezávislost na zdroji dat. Využití jediné implementace modelu ke zpracování různých typů vstupních dat nelze brát jako nevýhodu, naopak to může poskytnout velkou variabilitu systému, nicméně je nutné v modelu implementovat parsovací funkci pro rozlišení jednotlivých záznamů, a to pro každý typ dat. Pokud data nebudou dodržovat určité schéma, může být získávání záznamů velmi náročné, obzvlášť pokud je potřeba zpracovávat velmi objemná data. V další části práce o popisu vybraných implementací modelu jsou rozebrány datové modely systémů. Lze tedy vidět, že pro efektivní zpracování je dodržení určitého schématu nutnost.

Další možnou nevýhodou je implementace mapovací funkce a funkce redukce. Opět to může být na jednu stranu výhoda v podobě velké variability funkcionality, zejména při zpracování specifických nestrukturovaných dat. Na druhou stranu dotazovat data tímto způsobem nemusí být příliš efektivní v případě častého používání jednodušších dotazů, uživatel musí mít neustále na paměti styl map-reduce, tedy že mapovací funkce vytváří

dvojice typu klíč:hodnota, které jsou seskupeny ve funkci redukce. Uživatelem modelu je tedy většinou programátor. Pokud tedy srovnáme dotazování na data s dotazovacím jazykem SQL, je SQL uživatelsky příjemnější a nevyžaduje další znalosti algoritmizace funkcí v ramci MapReduce.

Třetí nevýhodou je přenos dat. Toto není problémem čistě MapReduce modelu, ale všech databází využívající během zpracování distribuované prostředí. Vždy je nutné při zpracování dotazu brát v potaz propustnost datové sítě, která ne vždy dosahuje dostačujících hodnot. V případě zpracování dat na jediném uzlu a udržování hodnot v operační paměti, jejichž propustnost dosahuje jednotek GB/s, je přístup ke zpracovávaným datům téměř okamžitý a přenos dat tak nebývá slabou stránkou systému. V případě distribuovaného systému se právě přenos dat stává slabým místem, obzvlášť u modelu MapReduce, kdy se z mapovacího uzlu na uzel redukce mohou v některých případech přenášet celé kolekce dat. Protože datové sítě zdaleka nedosahují propustnosti operační paměti, je při aplikaci modelu MapReduce důležité znát předpokládané objemy dat a použitou datovou síť, a zjistit tak, zda nebude výhodnější využít pouze jeden uzel, který pak zajistí efektivnější zpracování dotazů.

3 Dostupné implementace modelu

V průběhu deseti let od prezentace modelu vznikla celá řada databází, které vznikly na základě MapReduce nebo byly o tento typ zpracování obohaceny, ať už to jsou komerční nebo volně dostupné databáze. Detailní popis většiny z nich a jejich vlastností je mimo rámec této práce, proto jsou nadále popsány jen vybrané open-source databáze, které vznikaly v počátcích rozšiřování MapReduce. Od té doby procházely různými změnami a opravaní, díky čemuž jsou o něco víc rošířené než jiné databáze. Popíšeme si dva typy implementací, a sice Hadoop MapReduce jako framework umožňující jednodušší vývoj distribuovaných systému, tedy tak, jak byl model na počátku představen, a dále pak samotné distribuované databázové systémy, které používají určitý framework MapReduce nebo obsahují vlastní implementaci modelu - MongoDB [9], CouchDB [10], Hive [12], HBase [13] a Jaql [14].

Tyto implementace si rozebereme z pohledu MapReduce, nebudeme se tedy zabývat detaily, které v této chvíli nejsou pro nás důležité. Zaměříme se především na typ databáze z pohledu reprezentace dat (dokumentově orientovaná, sloupcově či řádkově orientovaná databáze apod.) a použití modelu MapReduce v databázi.

3.1 Hadoop MapReduce

Jako nejznámější implementaci jako softwarový framework je vhodné uvést *Hadoop MapReduce* [3][7]. Pojem *Hadoop* je název pro zastřešení projektů týkajících se problematiky distribuovaných systémů. Projekty spadají pod organizaci *Apache Software Foundation*[6] a jedním z nich je právě Hadoop MapReduce. Některé další projekty Hadoop jsou např.:

- **HDFS** - distribuovaný souborový systém,
- **Hadoop Common** - knihovny komponent a rozhraní pro podporu ostatních Hadoop projektů,
- **HBase** - distribuovaná sloupcově orientovaná databáze využívající HDFS a Hadoop MapReduce,
- **Hive** - distribuovaný datový sklad využívající HDFS a dotazovací jazyk podobný SQL (HiveQL), který je pak převeden do mapovacích funkcí a funkcí redukce nad frameworkem Hadoop MapReduce,
- **Pig** - prostředí pro zpracování velmi rozsáhlých kolekcí dat pomoci skriptovacího jazyka *Pig Latin* poskytující vyšší abstrakci a jednodušší řešení namísto programování mapovacích funkcí a funkcí redukce Hadoop frameworku, do kterých systém automaticky překládá z Pig Latin.

Samotný framework Hadoop MapReduce je napsán v Javě, avšak aplikace, které jej využívají, v Javě napsány být nemusí. Při použití *Hadoop Streaming* lze mapovací funkce a funkce redukce aplikaci zpřístupnit pomocí spustitelný soubor či skriptu a *Hadoop Pipes* umožňuje zápis funkcí v jazyce C++.

3.1.1 MapReduce job

Proces zpracování jednoho úkolu (rozdělení dat, mapovací funkce a funkce redukce, distribuce výsledků) se nazývá MapReduce job. Jaký je postup při tomto procesu již přibližně víme, podíváme se na něj však jiným pohledem, a sice pohledem Hadoop frameworku. V procesu identifikujeme čtyři nezávislé objekty:

- **JobClient**, který spouští úkol,
- **JobTracker** řídící celý průběh úkolu - uzel master,
- **TaskTracker** vykonávající přidělené úlohy - pracovní uzel,
- **HDFS** - distribuovaný souborový systém určen pro sdílení souborů s daty mezi uzly.

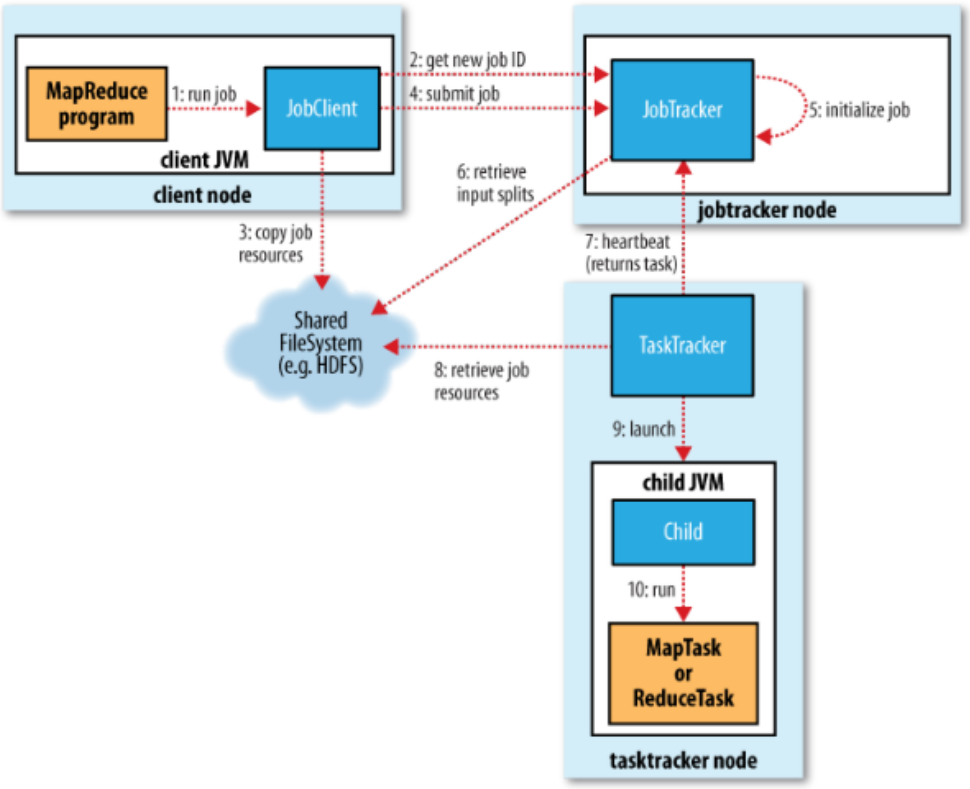
Nebudeme zde podrobněji popisovat jak tyto objekty mezi sebou spolupracují, princip je velmi podobný dříve popisovanému postupu, ale pro lepší představu jsou na obrázku 2 vyobrazeny jednotlivé kroky při spouštění úlohy. Nyní se spíš podíváme na distribuovaný souborový systém HDFS, který využívá většina projektů Hadoop.

3.1.2 HDFS

Distribuované souborové systémy jsou použity tehdy, pokud je potřeba transparentního pohledu na distribuovaná data. Systém může také automaticky vytvářet repliky na několika uzlech pro lepší dostupnost dat. Uživatel systému má pak distribuovanost dat skrytou a pracuje s nimi jako při použití běžného souborového systému. Pro tento druh problému byl pro potřeby projektů Hadoop vyvinut *Hadoop Distributed Filesystem* [3].

HDFS byl navržen tak, aby umožnil uchovávat především velké soubory, které mohou dosahovat i velikosti jednotek či desítek terabytů. Zpracování takto velkých souborů, kdy se celková velikost může vyšplhat na desítky petabytů, zahrnuje vytvoření rovněž takto velkých kolekcí výsledků. Proto se autoři drželi přístupu "zapiš jednou, čti víckrát". Systém je tedy určen pro systémy, které data nahrávají zřídka, ale o to víc se zaměřují na zpracování či jiné analýzy dat. Čas potřebný ke čtení celé kolekce je důležitější než latence při čtení jednotlivých záznamů. Umožňuje tedy streamování pro dosažení vysoké propustnosti. Další vlastnost HDFS je typická obecně pro MapReduce model, a sice fakt, že systém nevyžaduje vysoce spolehlivý a nákladný hardware, ale vystačí si s běžně dostupnými stroji, které jsou podstatně méně spolehlivé a šance na výskyt chyby uzlu je zde vysoká. HDFS je navržen tak, aby v případě chyby jednoho či více uzlů byl systém schopen i nadále pracovat samostatně a uživatel nepocítil nějaké podstatné přerušení.

Je logické, že existuje spoustu případů, kdy není vhodné HDFS použít. Jedním z nich je situace, která vyžaduje velmi nízkou latenci. Řekli jsme si, že systém je navržen pro čtení kompletních kolekcí. Při velké propustnosti běžné stroje nedosáhnou nízké latence. Také při velkém množství souborů systém není vhodný. Metadata o souborech a složkách se uchovávají v paměti, a při např. miliardě souborů je běžná paměť nedostačující.



Obrázek 2: MapReduce job v systému Hadoop (převzato z [3])

Další zajímavou vlastností je velikost bloku souborového systému. Disky mají běžně bloky velké 512 bytů a souborové systémy několik KB. HDFS má však bloky velké 64 nebo i 128 MB. Díky tomu můžou v systému existovat soubory o velikosti přesahující velikost jednoho disku, které budou rozloženy na několika uzlech.

3.2 Pig

Použití Hadoop MapReduce jako softwarového frameworku může být omezující v absenci jakéhokoliv uživatelského rozhraní a nutnosti implementace mapovací funkce a funkce redukce. Při složitějším zpracování se může kód těchto funkcí stát nepřehledný, málo znovupoužitelný nebo časově náročnější na vývoj.

Proto byl vytvořen dotazovací jazyk zvaný *Pig Latin* [8] zavádějící vyšší míru abstrakce nahrazující implementaci mapovací funkce a funkce redukce. K němu patřící systém *Pig* provádí překlad Pig Latin do mapovací funkce a funkce redukce (map-reduce jobs), které jsou následně spouštěny nad výše popisovaným Hadoop MapReduce. Pro ukázku si uvedeme jednoduchý příklad.

Mějme následující SQL dotaz, který nad tabulkou URLs (url, category, pagerank) vypíše průměrný pagerank kategorií s velkým počtem stránek (10^6), který mají pagerank vyšší než 0,2:

```
SELECT category, AVG(pagerank)
FROM URLs
WHERE pagerank > 0,2
GROUP BY category
```

a k němu ekvivaletní Pig Latin dotaz:

```
good_urls = FILTER urls BY pagerank > 0,2;
groups = GROUP good_urls BY category;
output = FOREACH groups GENERATE category, AVG(good_urls.pagerank);
```

Na tomto příkladě lze vidět, že Pig Latin používá pro transformace dat (filtrace, seskupení, agregační funkce, ...) velmi podobné klauzule jako SQL. Na druhou stranu se jedná o sled kroků, kdy každý z nich provádí jednu transformaci, podobně jako v klasickém programovacím jazyce. Rozpis dotazů do těchto kroků je rozhodně přehlednější a rychlejší než samotné programování dotazu. Jedná se v podstatě o spouštěcí plán, který umožňuje lehčí porozumění a řízení, jak jsou data v konkrétní úloze zpracována.

Pro spouštění Pig Latin dotazů se data do systému nijak nenahrávají nebo nedodržují předem definovaný formát, je tedy nutné implementovat parsovací funkci. Stejně tak si uživatel může zvolit formát výstupních dat. Dále má uživatel možnost sám definovat vlastní funkce, které potřebuje použít v dotazech pro svá specifická data. Implementace těchto funkcí je prováděna v jazyce Java, autoři ale slibují v budoucnu rozšíření do většiny nejpoužívanějších jazyků (C/C++, Perl, Python, apod.).

3.3 MongoDB

MongoDB je dokumentově-orientovaná databáze [9]. Záznamy jsou ukládány v *kolekcích* (collections - analogický název pro tabulky v RDBMS) v podobě tzv. *BSON* objektů. V rámci MongoDB nazýváme tyto objekty *dokumenty* (documents - analogický název pro záznam v RDBMS). BSON objekty jsou serializované *JSON* objekty (JavaScript Object Notation), což je jednoduchý textový formát pro ukládání a přenos dat skládající se z kolekce párů typu klíč:hodnota, přičemž hodnota může mít podobu pole [6]. MongoDB se dle autorů vyznačuje mimo jiné těmito vlastnosti: *lehkost* (lightweight) - díky serializaci JSON objektů nejsou dokumenty dat zbytečně velké co do velikosti na disku, což je vhodné zejména tehdy, když jsou přenášena po síti, dále a *dynamika* (schema free) - umožňuje změnu schéma dat, tzn. dva záznamy stejné kolekce mohou mít odlišné atributy. Na kolik to lze brát jako výhodu je nutné vyhodnotit z pohledu konkrétního použití.

Databáze MongoDB je napsána v C++, ale od uživatele vyžaduje určitou znalost JavaScriptu, např. pro administraci v shell nebo specifikaci mapovacích funkcí a funkcí redukce. Ekvivalent SQL dotazu uvedeného v popisu systému Pig by mohl vypadat následovně:

```
db.URLs.group({
    ”key”: {
        ”category”: true
    },
    ”initial”: {
        ”sumForPageRank”: 0,
        ”countForPageRank”: 0
    },
    ”reduce”: function(obj, prev) {
        prev.sumForPageRank += obj.pagerank;
        prev.countForPageRank++;
    },
    ”finalize”: function(prev) {
       prev.averagePageRank = prev.sumForPageRank / prev.countForPageRank;
        delete prev.sumForPageRank;
        delete prev.countForPageRank;
    },
    ”cond”: {
        ”pagerank”: {
            ”$gt”: 0.2
        }
    }
});
```

Lze si povšimnout, že mapovací funkce je v tomto dotazu zcela vynechána. Zároveň dotaz nepůsobí výrazně přehledně a je třeba se v něm zorientovat. Oproti Pig Latin je tento způsob dotazování více ”programátorsky”zaměřen. Zpracování pomocí MapReduce je v MongoDB využito jen při provádění agregačních dotazů.

V MongoDB je možno provádět vkládání a odkazování v dokumentech na jiný. Vkládání je ”zahnízdění”objektů a polí uvnitř dokumentu, zatímco odkazování je reference mezi dokumenty. MongoDB neprovádí spojování kolekcí (tabulek) kvůli možné vysoké náročnosti v rozsáhlém distribuovaném prostředí. Obecně se doporučuje používat vkládání tam, kde je potřeba vyjádřit vztah mezi entitami. Vkládání je tedy určitou formou částečného spojení kolekcí. Naproti tomu, odkazy je možné použít tam, kde nechceme aby nám vznikala případná duplikace dat, avšak práci s těmito odkazy je třeba implementovat v klientské aplikaci formou dalších dotazů.

Dokumenty kolekcí nemají pevně danou strukturu. Dokonce i v rámci jedné kolekce mohou mít dokumenty strukturu rozdílnou. Toto se však příliš nedoporučuje, v praxi je výhodnější mít kolekci homogenní, kdy struktura se řídí konceptuálním návrhem pro konkrétní databázi. Tato vlastnost může být brána jako nevýhoda, obzvlášť, pokud pracujeme ve velkém týmu. Struktura zkrátka není striktně dána a MongoDB ji implicitně nekontroluje a v evidenci dat pak mohou nastat nějaké nesrovnalosti. Na druhou stranu, máme velkou volnost, a v situacích, kdy je nutné přidat jeden či více atributů, nemusíme

upravovat strukturu celé kolekce, jako provádí v SQL příkaz "ALTER TABLE". Data v databázi můžeme tedy zařadit do skupiny semi-strukturovaných dat.

3.4 CouchDB

Další distribuovaný databázový systém, který stejně jako Hadoop spadá pod licenci Apache, zmíníme CouchDB [10]. Systém je napsán v jazyce Erlang - jazyk vyvinutý pro vývoj distribuovaných systémů požadující vysokou škálovatelnost, dostupnost a odolnost vůči chybám [11]. CouchDB a MongoDB jsou často porovnávány, neboť mají společné rysy, např. obě jsou dokumentově orientované, při použití MapReduce specifikují funkce v JavaScriptu, data jsou semi-strukturovaná - i CouchDB používá JSON. Dotazování je tedy velmi podobné jak u MongoDB.

Co se týče MapReduce, zde se s ním setkáme ve větší míře než jen v případech dávkového zpracování či použití agregačních funkcí - CouchDB používá MapReduce pro sestavení každého pohledu (views). Toto může být nevýhoda pro programátora, který je zvyklý na získávání dat pomocí klasických dotazů. Autoři systému však předpokládají, že se nám to vrátí rychlejší odezvou díky výkonu MapReduce a lehkosti jazyka Erlang. Pro specifikaci funkcí v jiných jazycích, než je Erlang, je možné použít doplňky třetích stran.

3.5 Hive

Hive je jedním z podprojektů Hadoop [12]. Jedná se o databázi v podobě datového skladu využívající souborový systém HDFS a implementaci Hadoop MapReduce. Stejně jako předchozí uváděné systémy, i Hive poskytuje usnadňuje dotazování dat pomocí skriptovacího jazyka, a size pomocí *HiveQL* - dotazovací jazyk podobný SQL. Popišme si tedy tento typ databáze trochu podrobněji.

V Hive jsou data strukturovaná a jsou organizovány do tabulek, oddílů a bloků.

Tabulky jsou analogické k tabulkám v relační databázi, kdy každé tabulce je v systému HDFS přidělen jeden adresář, ve kterém se v souborech o dané velikosti nacházejí serializovaná data. Hive nabízí podporu primitivních datových typů - integer, float, string, date, boolean; kolekce typu map a array. Rovněž je zde možnost definice vlastních datových typů, k těm se však musí implementovat funkce pro serializaci a deserializaci dat.

V základním adresáři tabulky se mohou vyskytovat jeden nebo více podadresářů - oddílů. *Oddíly* se vytvářejí nad jednotlivými sloupci, tedy pokud máme např. databázi obchodu s tabulkou "prodej", kterou chceme rozdělit dle data prodeje, záznamy o prodeji zboží dne 1.5.2011 se budou nacházet v souboru v adresáři např. /obchod/prodej/datum=01052010. V těchto podadresářích se mohou vyskytovat další, pokud požadujeme oddíly nad více sloupci.

Data v oddílech jsou pak dělena do *bloků* jehož velikost je odvozena z hash klíče daného sloupce tabulky. Každý blok má na disku podobu jednoho souboru v odpovídajícím adresáři.

Jak bylo zmíněno, určitého usnadnění použití modelu MapReduce je dosaženo pomocí dotazovacího jazyka HiveQL (někdy také jen HQL). Z dotazu jsou při kompilaci vytvořeny ekvivalentní mapovací funkce a funkce redukce.

HiveQL pro čtení dat nabízí standartní klauzule jako SQL - selekce, projekce, spojování tabulek, agregační funkce nebo poddotazy; dále poskytuje syntaxi pro vytváření tabulek a specifikaci formátu atributů (DDL - Data Definition Language), syntaxi pro nahrávání dat do databáze (DML - Data Manipulation Language), ale již nepodporuje editace či mazání záznamů. Lze odstranit jen celý oddíl nebo tabulku a při editaci se přepisuje celý oddíl. Vzhledem k tomu, že Hive je vytvořen především jako datový sklad a data se zpravidla aktualizují či vkládají v určitých intervalech (dny, hodiny), zmíněné omezení editaci či mazání nejsou velký problém - stavající data se nahrají do nového oddílu či tabulky.

Systém také umožňuje rozšiřovat funkcionalitu implementací vlastních funkcí v různých programovacích jazycích pro úpravu atributu záznamu (UDF - user-defined functions) a agregační funkce (UDAF - user-defined aggregation functions), případně implementovat samotné funkce mapovací funkce a funkce redukce.

Při dotazování se výsledky automaticky vkládají do nové tabulky. Dotaz je tedy rozšířen o specifikaci tabulky, v případě použití Hive CLI tabulky není třeba specifikovat, postará se o to systém. Jednoduchý dotaz pro výběr aktivních uživatelů může tedy vypadat takto:

```
INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;
```

Kromě prvního řádku, kde určujeme název nově vytvořené tabulky s výsledky, je dotaz totožný s SQL. Další dva jednoduché příklady jen potvrzují, že uživatelé znalí SQL nebudou mít téměř žádné problémy pracovat se systémem Hive. Pro potřeby systému je HiveQL rozšířen o některé klauzule, vždy ale můžeme vycházet z toho, že je podobný SQL. Výše uváděný dotaz pro výpočet průměrného pageranku dle kategorií by vypadal kromě prvního řádku naprosto stejně jako zápis v SQL:

```
INSERT OVERWRITE TABLE pagerank_by_category
SELECT category, AVG(pagerank)
FROM URLs WHERE pagerank > 0.2
GROUP BY category
```

3.6 HBase

Další databází, na kterou se podíváme a rovněž patří pod Hadoop, je HBase [13]. Je to další distribuovaný úložný systém pro velmi rozsáhlé kolekce semi-strukturovaných dat. Pro tento systém je specifická inspirovanost sloupcově orientovanými systémy. Pro ukládání souborů se nejčastěji používá souborový systém HDFS, architektura HBase však dovoluje použít i jiný, uživatelem zvolený, systém, který ovšem nemusí mít všechny požadované vlastnosti a možnosti systému mohou být lehce omezeny.

V běžných řádkově orientovaných systémech jsou všechny hodnoty atributů jednoho řádku (záznamu) souvisle zapsány na disk. V případě sloupcově orientovaných systémů jsou za sebou souvisle zapsány hodnoty sloupce. Tento druh databáze se používá v případech, kdy systém provádí dotazy, které ve většině případů nevyžadují všechny nebo většinu atributů jednotlivých záznamů. Provádění agregačních funkcí (např. MAX(), AVG(), COUNT(), apod.) je pak rychlejší než u klasické řádkově-orientované databáze. Databáze sloužící pro analytické účely velmi často využívají tento druh databáze. Další důvod k užití této databáze je komprese dat - hodnoty sloupců jsou často podobné, a proto mohou být efektivněji komprimovány, narozdíl od komprese řádků, kdy hodnoty bývají značně rozdílné.

V různých zdrojích je HBase uváděna právě jako tento typ databáze, nicméně autoři uvádějí, že mezi ně nepatří - jen se inspiruje způsobem ukládání dat na disku. Hlavní rozdíl je, že sloupcově orientované systémy nabízí efektivní analytický přístup k datům, zatímco HBase vyniká v rychlosti přístupu ke konkrétní buňce (hodnotě atributu jednoho záznamu) pomocí specifického klíče, případně ke zvolenému rozsahu buněk.

Pro provádění map-reduce úloh je nutné implementovat mapovací funkce a funkce redukce v jazyce Java. HBase tedy původně neobsahuje vlastní nebo jiný dotazovací jazyk.

HBase data organizuje do řádků a sloupců, respektive tabulek, přičemž sloupce mohou mít více verzí - každá hodnota ve sloupci má svou časovou známku. Na konkrétním řádku a konkrétním sloupci se proto může vyskytovat více buněk různého typu nebo jsou stejného typu a při čtení hodnoty se čte nejprve ta nejnovější.

Řádky jsou lexikograficky seřazeny dle klíče, který je unikátní pro celou tabulku. Můžeme si jej proto představit jako indexovaný primární klíč v klasických relačních databázích. HBase dále povoluje *sekundární klíče*.

Řádky samozřejmě sestávají z hodnot sloupců, které se seskupují do tzv. *rodin* (column family). Takto lze jasně rozdělit sloupce do skupin např. podle jejich významu či jiné podobnosti. Sloupce ve stejné rodině jsou pak společně zapsány na disk do souboru, který se v rámci HBase nazývá *HFile*. Rodiny se definují při vytváření tabulky a po vytvoření již nemohou být změněny. Také se doporučuje vytvářet maximálně pár desítek rodin v jedné tabulce. Na druhé straně, počet sloupců v tabulce není nijak omezen, dokonce není omezena velikost a typ hodnot ve sloupcích.

3.7 Jaql

U Hadoop nadále zůstaneme a popíšeme si jazyk *Jaql* [14]. Jaql, od vývojářu z IBM, je skriptovací jazyk pro Hadoop MapReduce, a je použit v několika komerčních produktech společnosti. Samozřejmě, i v případě jazyku Jaql byl jeho vývoj podmíněn snahou usnadnit a urychlit práci v prostředí MapReduce, respektive usnadnit vytváření analýz semistrukturovaných dat v paralelním prostředí. Jaql se skládá ze tří hlavních částí, a sice ze skriptovacího jazyku, kompilátoru a spouštěcího prostředí pro Hadoop.

Datový model je jednoduchý - hodnota(value) je vždy buď *atom*, *pole* nebo *záznam*. Atom může nést hodnoty všech základních datových typů, včetně typů jako jsou string, date či number. Pole jsou setříděné kolekce hodnot (tzn. nejen atomů, ale i jiných polí či záznamů) zatímco záznamy jsou nesetříděné kolekce nám známých dvojic typu klíč:hodnota.

Navzdory jednoduchosti je datový model flexibilní a umožňuje tak pracovat s různou reprezentací dat, např. s textovými, binárními či JSON soubory, relačními databázemi, nebo XML dokumenty.

Jaql je skriptovací jazyk. Skript tvoří sekvence příkazů, kdy každý příkaz je vždy buď *import*, *assignment*(přiřazení) nebo *expression*(výraz). V [14] je uveden následující jednoduchý skript, sloužící jako dobrý příklad pro seznámení se s Jaql jazykem. Příklad má zjistit, které dvojice klíč:hodnota se v datech nacházejí a jejich počet.

```
import myrecord;

countFields = fn(records) (
  records
  -> transform myrecord::names($)
  -> expand
  -> group by fName = $ as occurences
    into {name: fName, num: count(occurrences)}
);

read(hdfs("docs.dat"))
  -> countFields()
  -> write(hdfs("fields.dat"));
```

Samotné zpracování souboru s daty je provedeno posledními třemi řádky, tedy načtení souboru ze souborového systému hdfs, provedení funkce countFields, a zápis výsledku do souboru. Tato funkce závisí na externě definované funkci, která je importována na řádku 1 z modulu "myrecord".

Oproti jazyku Pig Latin, ve kterém byla při každém novém kroku vytvořena nová proměnná, je zde použita syntaxe "$->$". Označuje tok dat, respektive tok výstupu jednoho výrazu na vstup výrazu dalšího. Dále výraz *transform* aplikuje na každý prvek vstupního pole funkci, a vytváří tak pole nové. V uvedeném příkladu je pole "records"podstoupeno funkci "names"z modulu "myrecord". Znak '$' zajistí provedení funkce na každý prvek vstupního pole. *expand* pak provede sjednocení polí do jednoho, respektive provede odebrání jedné vrstvy zanoření hodnot v poli. Nakonec výraz *group by*, velmi podobný GROUP BY z dotazovacího jazyku SQL, seskupí prvky pole na základě výrazu a aplikuje agregační funkcí, v našem případě "count", na každou skupinu prvků.

Jazyk Jaql obsahuje mnoho dalších klíčových slov, např. *join*, *filter* či *union*, ale cílem této práce není podrobný popis tohoto jazyka. Jako většinu jiných vysokoúrovňových jazyků pro práci s modelem MapReduce lze i tento dále rozšiřovat pomocí vlastních funkcí (UDF - user-defined functions). Ty lze zapisovat jak v jazyce Jaql tak i v jiných programovacích jazycích.

3.8 Další systémy a shrnutí

Dalších implementací modelu MapReduce existuje celá řada, ať se jedná o databázové systémy nebo datové sklady s vlastní implementací modelu, či jazyky vyšší úrovně pro práci

s existující implementací modelu MapReduce. Kromě systémů Jaql patří výše popsané mezi nejčastěji používané open-source implementace.

I na malém množství systémů, které byly v předchozích podkapitolách stručně popsány, lze vidět určitý vývoj, kdy se systémy z "čisté"implementace modelu MapReduce postupně transformovaly v dokonalejší systémy a to především na z těchto pohledů:

1. vlastní datový model
2. dotazovací jazyk

Dotazování pomocí jazyka SQL nebo jiného je rozhodně jednodušší než samotné programování mapovacích funkcí a funkcí redukce. Výhodou zůstává možnost doimplementovat vlastní funkce. To samé platí i pro přítomnost vlastního datového modelu - není třeba implementace vlastní parsovací funkce. Systém pak využívá vlastní optimalizované metody pro načítaní dat, díky čemuž je zpracování efektivnější. Systémy se tedy snaží odstranit nevýhody výplývající z modelu, které jsme si popsaly v předchozí kapitole.

V Google používali pro jednodušší práci s MapReduce vlastní jazyk *Sawzall* [15], který je podobný jazyku Pig nebo Jaql. Sawzall je dostupný jako open-source, ale jen bez podpory MapReduce, proto nemá cenu se jim blíže zabývat. Sawzall byl postupně nahrazen systémem *Tenzing* [16], což je platforma pro překlad rozsáhlé části SQL (SQL92, z části i SQL99) do kódu MapReduce, který je v Google postaven na distribuovaném souborovém systému GFS. Dalšími systémy, které umožňují jednodušší práci s modelem jsou např. *YSmart* [17] nebo *HadoopDB* [2]. YSmart používající SQL má mít dle tvůrců rychlejší a optimalizovanější provádění map-reduce úloh. V práci [17] se zaměřili na problémová místa systémů Pig nebo Hive, které rovněž provádějí převod vysokoúrovňového jazyka do MapReduce. HadoopDB kombinuje výhody paralelních distribuovaných systémů a modelu MapReduce. Je open-source, je založen na Hadoop MapReduce a databázi PostgreSQL a jako jazyk pro práci s MapReduce využívá HiveQL s mírným rozšířením.

Hadoop MapReduce se stal základem pro většinu projektů, které různým způsobem pracují s modelem MapReduce. Ostatní open-source frameworky jsou buď založené právě na Hadoop MapReduce a určitým způsobem zjednodušují práci nebo jiné nejsou tak rozšířené a jejich podpora a dokumentace není na dobré úrovni. Důležitější však je důkladně analyzovat záměr distribuovaného systému a vhodnost použití MapReduce (požadovaná dostupnost, rychlost, škálovatelnost, míra konzistence, apod.). Například v práci [2] porovnávali výkon frameworku Hadoop MapReduce, DBMS-X - paralelní relační databáze a systému Vertica - sloupcově orientovaný datový sklad. DBMS-X a Vertica neimplementují MapReduce a dotazy nad daty provádí pomocí SQL. Ve většině testů v podobě různých typů úloh byl Hadoop vyhodnocen jako nejpomalejší. Komerční produkty ovšem nabízí systémy s "odladěným"Hadoop MapReduce, u nichž deklarují několikrát větší rychlost, a také hodně záleží na zkušenosti programátora map-reduce úloh.

Další dostupné implementace jsou např.:

- **DisCo** - framework založen na Hadoop psán v jazyce Python [33],
- **Skynet** - vlastní implementace MapReduce v jazyce Ruby [34],

- **Phoenix++** - implementace opět pro prostředí se sdílenou pamětí (shared-memory) poskytující API v jazyce C či C++ [35].

Co se týče komerčních produktů, mezi ty nejvýznamnější se dají zařadit GreenPlum Database [36], Teradata Aster MapReduce [37] a Amazon Elastic MapReduce [38].

GreenPlum Database je systém pro uchování, řízení a analýzu rozsáhlých dat. Autoři jej označují jako datový sklad nové generace, který kombinuje paralelní zpracování SQL a MapReduce. Přímo podporuje také zpracování dat z Hadoop clusteru.

Teradata Aster MapReduce rovněž nabízí výhody jednoduchosti SQL v kombinaci s efektivitou MapReduce, a stejně jako GreenPlum Database je určen především pro uchování a analýzu velmi rozsáhlých dat.

Amazon Elastic MapReduce je webová služba poskytována společností Amazon na svých cloudových službách virtuálních strojů a úložných prostorů - Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3). Na nich běží Hadoop framework, který provádí zpracování dat.

Dále existují společnosti jako MapR Technologies nebo Cloudera poskytující systémy kombinující framework Hadoop MapReduce s jinými systémy Hadoop (Hive, HBase, Pig, apod.). Tyto systémy jsou upraveny tak, aby poskytovaly jednodušší práci a rychlejší zpracování.

Takže i když se bavíme o komerčních produktech, můžeme říci, že mnoho z nich také stojí na projektech Apache Hadoop. Hadoop, open-source implementace modelu, se tedy stal určitým základem a impulsem pro vývoj systémů využívající popisovaný model.

Nemá cenu se dále zabývat detailním pohledem na každý systém implementující model MapReduce, neboť existují i další než výše popsané a jejich podrobnější popis by přesahoval rámec této práce. Udělali jsme si přehled, co současný trh na poli zpracování rozsáhlých dat nabízí, nyní je vhodné vzít si z těchto poznatků příklad pro následující návrh implementace modelu.

4 Analýza a návrh implementace modelu MapReduce

Jedním z cílů této práce je i implementace distribuovaného systému, který pro zpracování rozsáhlých dat využívá popisovaný model MapReduce. V implementaci je využito znalostí nabytých studiem modelu a jeho některých stávajících implementací z předchozí kapitoly práce.

4.1 Analýza požadavků na implementaci

Základní požadavky na implementaci lze shrnout do několika bodů:

1. nezávislost na velikosti dat
2. použití dotazovacího jazyka
3. sekvenční čtení záznamů
4. jednoduchá implementace fyzického uložení dat
5. minimalizace přístupu na disk
6. použití v distribuovaném prostředí
7. využití plné replikace
8. implementace v C++, vhodné řešení správy paměti

Jedním z hlavních požadavků na implementovaný systém je schopnost za pomocí modelu MapReduce zpracování rozsáhlých dat, řádově jednotky či desítky GB. Na tento požadavek se váže i požadavek na efektivitu samotného zpracování, data by se tedy měla zpracovávat sekvenčně. Nezávislost na velikosti dat bude nutné brát v potaz v každé části implementace, je proto potřeba vhodně volit použité datové struktury a algoritmus zpracování.

V předchozí kapitole jsme zmínili jednu z možných nevýhod modelu, a sice nutnost implementace vlastní mapovací funkce a funkce redukce pro dotazování dat. Tuto nevýhodu některé systémy odstranili zavedením vlastního dotazovacího jazyku. Vlastní dotazovací jazyk je požadavek i pro implementovaný systém. Vzhledem k tomu, že každý z vlastních vyvinutých dotazovacích jazyků lze převést na ekvivaletní SQL dotaz, je dotazovací jazyk SQL vybrán jako nejvýhodnější pro použití v implementovaném systému. Rozsáhlost SQL je velmi široká, proto se zaměříme jen na jeho malou podmnožinu, která nám poskytne funkcionalitu nejnutnější k demonstraci funkční implementace modelu.

Rovněž se budeme snažit eliminovat další vlastnost modelu, která může být považována za nevýhodu, a to univerzálnost na formátu vstupních dat (např. Hadoop MapReduce). Bylo popsáno, že v modelu je nutné implementovat vlastní parsovací funkci, která "zpřístupní"jednotlivé záznamy. Tato činnost může být časově náročná. Použití vhodného formátu uložených dat se znalostí schématu je dalším požadavkem na systém,

který by měl umožnit rychlejší načítání a zpracování dotazu než parsování záznamů a vyhodnocení datových typů atributů až po spuštění dotazu.

Jelikož je model uplatňován především v distribuovaném prostředí, je distribuované zpracování jedním z dalších požadavků na implementaci. V síti uzlů je nutné při spuštění dotazu vytvořit logickou hierarchii sestávající z master uzlu a pracovních uzlů podle popisu modelu v teoretické části. Je nutné zajistit vzájemnou znalost serverů tak, aby přenášení mezivýsledků probíhalo přímo mezi pracovními uzly. V modelu má své využití i replikace dat pro výkonnější zpracování, proto se v systému zaměříme na to, aby při plné replikaci dat (každý uzel obsahuje stejnou kopii dat) bylo využito maximální možné paralelizace a výpočetního výkonu všemi uzly. Plná replikace však není v praxi vždy využívána, v modelu MapReduce je replikace nejčastěji trojnásobná, jedná se však o využití v clusteru desítek či stovek uzlů. V našem případě není předpoklad pro tak vysoký počet uzlů, proto si vystačíme s plnou replikací, díky čemuž nebude třeba řešit správu kopií na jednotlivých uzlech a zároveň bude tento typ replikace pro testovací účely vhodnější.

Model MapReduce je svým původem určen pro využití na levném hardwaru, nepředpokládá se časté využití výkonných serverů. Existují modifikace modelu, které výkonné servery pro zpracování vyžadují, ale my se zaměříme na nezávislost na použitém hardware. S tím souvisí omezená kapacita operační paměti a pevných disků. Pro co největší časovou efektivitu je výhodnější vyhnout se přístupu na disk. Přístup na disk je časově drahá operace a proto je vhodné ji co nejvíce eliminovat. Vzhledem k požadavku na schopnost zpracování velmi rozsáhlých dat a omezené operační paměti bude nutné data zpracovávat po částech. Jako vhodné dělení do částí se jeví rovnoměrné rozdělení do bloků o stejné velikosti, jak je to využito i u jiných, výše popsaných, implementací. S požadavkem na nezávislost použitého hardwaru může nastat situace, kdy jeden uzel je výrazně rychlejší než druhý. Podle popisu modelu v teoretické části víme, že jedna z výhod modelu je právě rovnoměrné rozložení zátěže na uzly podle výkonu. Čím větší datové bloky budou, tím víc nerovnoměrně se zátěž rozděluje. Proto je vhodné zjistit optimální velikost bloku. Tímto rozdělováním se však budeme zabývat jen okrajově, protože není v možnostech této práce testovat implementaci na desítkách či více uzlech s rozdílným výkonem. Testování proběhne na dvou serverech se spuštěnými několika instancemi aplikace, viz kapitola o testovacím prostředí.

Celý systém bude implementován jako konzolová aplikace v jazyce C++ [22], který umožňuje vlastní správu paměti na rozdíl od jiných jazyků, jako je Java nebo C#, u kterých je správa paměti řešena automaticky pomocí *Garbage Collectoru*. Ta je vzhledem k objemu zpracovávaných dat velmi podstatná a je důležité zvážit její použití, neboť tak, jako u omezení přístupu k pevnému disku, tak u operační paměti je vhodné zamezit její častou alokaci. Při časté alokaci dochází k časové ztrátě a také k fragmentaci paměti, což pak vede k nemožnosti alokovat souvislý blok o požadované velikosti. Aplikaci je pak možné spustit v režimu server, kdy se zprovozní za účelem připojení uzlu do clusteru, nebo v režimu klient sloužící pro samotné dotazování dat.

4.2 Návrh implementace na základě analýzy

Práce je zaměřena na zpracování dat v distribuovaném prostředí. Z toho důvodu je nutné zajistit komunikaci mezi jednotlivými uzly v síti. Pro implementovaný model není vyžadována konkrétní fyzická topologie sítě. Ve chvíli, kdy v map-reduce zpracování dochází k rozesílání dat z mapovacích uzlů na uzly redukce, by byla z hlediska výkonu nejvýhodnější polygonální topologie, tedy projení "každého uzlu s každým", nicméně není to nutnost a ani to model nepředpokládá. Vzájemně jsou si uzly rovnocenné do té doby, než je započata map-reduce úloha. Po dobu jedné úlohy vzniká jednoduchá, výše popisovaná, hierarchická struktura, kde je jeden uzel zvolen jako master a osatní jako pracovní. Podmínka je, aby se každý uzel mohl stát master uzlem.

K realizaci takového prostředí je využito stávající implementace distribuované datové struktury pro masivně paralelní zpracování dat[18], která byla po dohodě poskytnuta vedoucím této práce jako framework, který zajišťuje samotnou komunikaci mezi uzly. Jeho využití nám umožní lépe se soutředit na samotnou implementaci modelu a neřešit tak další problémy týkající se přenosu dat pomocí protokolu TCP/IP. Implementací vlastního modelu MapReduce je pak tento framework řízen.

Aby bylo dosaženo zmiňované rovnocennosti uzlů a funkční implementace modelu, je nutné, aby každý z uzlů znal adresy ostatních. Proto je nutné uzel při připojení do clusteru "konfigurovat"přidáním textového csv souboru se seznamem dostupných serverů. Vzhledem k vlastnostem modelu není nutné řešit prioritu serverů. Z tohoto důvodu a také důvodu jednoduché a rychlé možnosti editace bylo zvoleno umístit tabulku serverů do textového souboru. Tabulka je načtena vždy po spuštění aplikace. Soubor má název *S_SERVERLIST.txt* a tabulka obsahuje čtyři atributy: ID, NAME, IP, PORT. Tento způsob uchování adres je pro tuto práci dostačující. Ukázka, jak může tabulka vypadat, je uveden v tabulce 1.

ID	NAME	IP	PORT
0	server_0	192.168.1.1	40000
1	server_1	192.168.1.2	40000
2	server_2	192.168.1.3	40000
3	server_3	192.168.1.4	40000

Tabulka 1: Ukázka seznamu serverů

Jako nejvhodnější dotazovací jazyk byl pro svou známost a obsáhlost zvolen SQL[19]. Systém tedy dle zvoleného dotazu upraví průběh zpracování tak, aby bylo dodrženo zpracování dle modelu. V rámci práce však není možné implementovat celý jazky SQL, proto je implementována jen jeho malá podmnožina, která je dostačující pro základní dotazování a demonstraci modelu MapReduce. Jedná se o projekci vybraných atributů, selekci dle specifikované podmínky, která je vzhledem objemu testovaných dat nutná, provádění základních agregačních funkcí a jejich případné seskupení dle zvolených atributů. Rovněž se nebudeme zabývat problematikou operace *join*, jejíž důkladné řešení by přesahovalo rámec této práce, navíc řešení spojování u nerelačních databází nemá vzhledem k efektivitě příliš velký význam. Dále vzhledem k velkému předpokládanému objemu dat

nebude implementace zaměřena na vkládání či mazání jednotlivých záznamů. Byla tedy vybrána tato funkcionalita:

- **SELECT** - projekce dle požadovaných atributů s možností agregační funkce
- **FROM** - výběr tabulky
- **WHERE** - selekce záznamů dle specifikované podmínky
- **GROUP BY** - seskupení dle požadovaných atributů při použití agregační funkce

Syntaxe implementované podmnožiny SQL je následující:

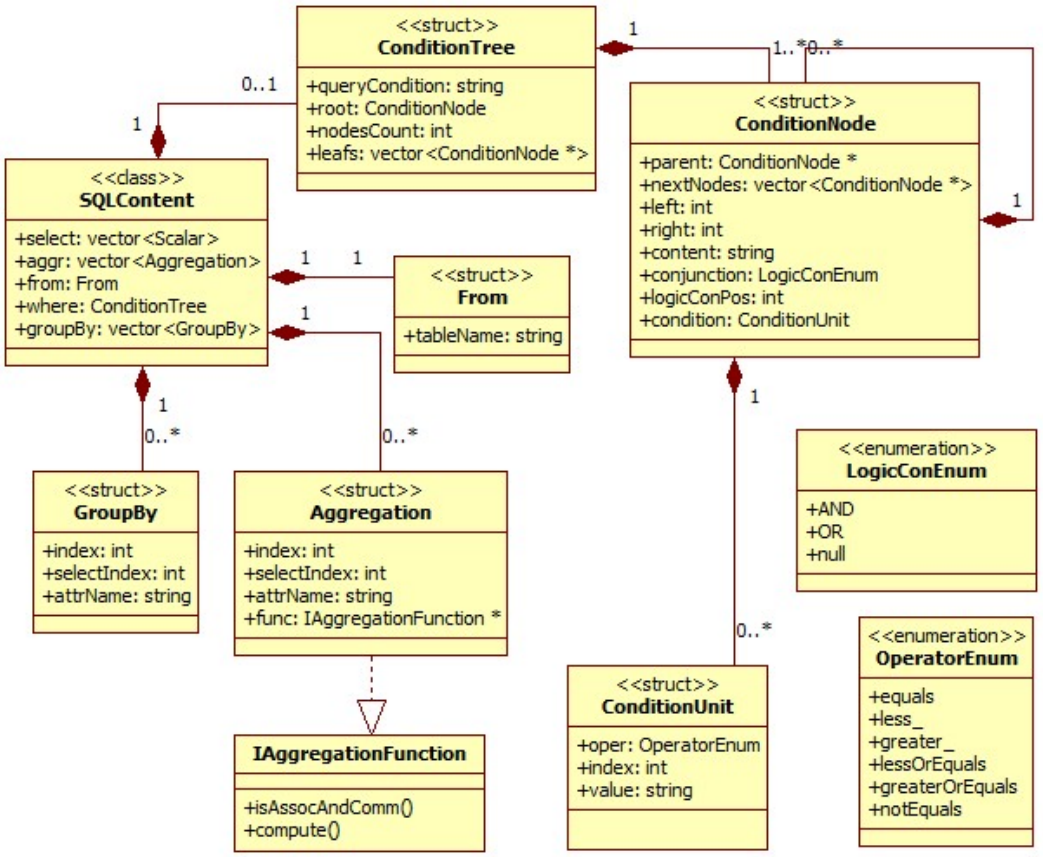
```
SELECT <atribut [, ...]  | *>
FROM <tabulka>
[WHERE <podminka [AND | OR <...>]>]
[GROUP BY <atribut[, ...]> ]
```

kdy atribut značí projekci vybraných atributů, respektive výběr všech atributů při použití znaku '*'. V případě numerického atributu lze na něj aplikovat jednu ze základních agreagační funkcí *MIN*, *MAX*, *SUM*, *AVG* či *COUNT*. Podmínku lze specifikovat pomocí operátorů =, <, >, <=, >= nebo <> s hodnotou odpovídajícího datového typu atributu, tedy s číselnou hodnotou, textového řetězce v uvozovkách nebo data ve formátu *'yyyy-MM-dd hh:mm:ss'*.

Návrh struktury reprezentující SQL dotaz během zpracování dotazu je popsána obrázkem 2 se zjednodušeným, ale dostatečně popisujícím, třídním diagramem struktury.

Vzhledem k možné velké časové náročnosti při parsování hodnot a převodu na vhodné datové typy při vykonávání agregační funkce je mezi dalšími požadavky použití vhodného formátu dat. Systém tedy nebude univerzální vzhledem ke vstupním datům, odpadne však nutnost implementace parsovací funkce pro každý typ dat a získáme efektivnější přístup k jednotlivým záznamům. Data budou reprezentovány tabulkami s řádky a sloupci, respektive se záznamy a jejich atributy. V systému jsou implementovány základní datové typy, kterých mohou atributy záznamů nabývat. Jsou to:

- *INTEGER* - celé číslo reprezentováno C++ datovým typem *int*,
- *DOUBLE* - reálné číslo reprezentováno C++ datovým typem *double*,
- *VARCHAR* - řetězec libovolné délky reprezentován datovým typem *char **.
- *DATETIME* - struktura pro reprezentaci času *struct DateTime* s využitím datového typu *unsigned short* pro hodnotu roku, pro hodiny a menší hodnoty je použit datový typ *unsigned char*, viz výpis kódu 1.



Obrázek 3: Třídní diagram návrhu reprezentace SQL dotazu pro zpracování.

```
struct DateTime {
    unsigned short year;
    unsigned char month;
    unsigned char day;
    unsigned char hour;
    unsigned char minute;
    unsigned char second;
}
```

Výpis 1: Reprezentace datového typu DateTime

Pro perzistenci veškerých dat je v systému použita datová struktura pole. Nejhorší možná složitost pro vyhledání záznamu je tedy $O(N)$, kde N je počet záznamů v poli. Veškeré záznamy jsou binárně uloženy v poli bajtů, přičemž při použití datového typu VARCHAR, kdy délka hodnoty může být u záznamů různá, se může lišit bajtová délka jednotlivých záznamů. Proto je použita i indexace záznamů, kdy index určuje pozici

počátku záznamu v poli. Pak přístup k záznamu podle jeho známého pořadí v datovém bloku probíhá v konstantním čase.

V databázových systémech je fyzická implementace uložení dat řešena do tzv. stránek, tedy bloků o stejné velikosti, které mohou být uloženy často v podobě B-stromu [21] s nejhorší složitostí vyhledávání $O(log_n(N))$, kde N je počet záznamů a n počet klíčů uzlu stromu. Díky nižší složitosti je zamezeno častému přístupu na disk, který představuje časově drahou operaci. Tento sofistikovanější způsob fyzické implementace dat je řešen především v relačních databází. V implementaci nám však postačí struktura pole, která s využitím indexů zajistí postačující složitost, a díky předpokladu načtení kompletního datového bloku do operační paměti zamezíme častému přístupu na disk.

Tabulky jsou v systému vytvořeny až importem dat. Při importu dat probíhá parsování csv souboru se záznamy a rozeznávání datových typu dle přiloženého souboru se schématem. Při importu je zvolena maximální velikost datového bloku tabulky, což je dáno požadavkem na zpracování rozsáhlých kolekcí dat a omezenou operační pamětí. Při zpracování je pak v paměti nahrán jen jeden blok. Pokud chceme zamezit častému přístupu na pevný disk, je vhodné volit velikost bloku tak, aby byl vyvážen poměr mezi kapacitou operační paměti a efektivitě zpracování. Nejčastěji proto budeme volit velikost bloku v rozmezí 128 MB - 512 MB. Během importu jsou vytvořeny datové struktury popisující tabulku a záznamy, které si následně popíšeme. Tyto struktury jsou během importu ukládány na disk a během vykonávání dotazu jsou pak všechny (pro jeden datový blok) načteny v paměti. Vytvářené soubory/struktury jsou popsány v následujících odstavcích.

Následuje popis návrhu samotného vykonání dotazu. Při distribuovaném vykonání dotazu vzhledem k map-reduce je nutné se držet rozdělování úkolů a dat pro uzly popsané v teoretické části práce. V této podkapitole je postupně popsáno zpracování dotazu podobně jako v teoretické části práce, ale v rámci implementace.

4.2.1 Spuštění dotazu

Klient zasílá dotaz na vybraný server z tabulky serverů. Po přijmu dotazu na serveru proběhne jeho formální kontrola zda je SQL dotaz zapsán korektně a také kontrola oproti schématu zvolené tabulky. Pokud kontrola proběhne v pořádku, server se v rámci úlohy stává masterem a řídí průběh zpracování. V popisu modelu dle [1] v této chvíli je master zodpovědný za rozeslání celkem M mapovacích úloh, kdy M je počet bloků stejné velikosti, na které jsou vstupní data rozdělena. V implementaci je toto řešeno v podstatě stejně až na to, že se pracovním uzlům nebudou zasílat úlohy po jedné, ale celá dávka úloh. Při odesílání úlohy je každému mapovacímu uzlu přiřazeno číslo značící jeho pořadí mezi ostatními mapovacími uzly. Dle tohoto pořadí pak uzel ví, které bloky dat má v případě replikace zpracovávat. Bloky dat jsou tak rovnoměrně rozděleny mezi uzly. Pokud je spuštěn dotaz nad nereplikovanými daty, hodnota pořadí uzlu nehraje pro samotný uzel roli, neboť je nutné zpracovat všechny bloky dat na uzlu.

O veškerou serverovou komunikaci se stará třída *MRServer*. V hlavičce přijaté zprávy se vždy předává hodnota, díky které server rozezná, zda má pracovat v režimu Master, Map uzel či uzel redukce. Podle toho jsou pak spouštěny požadované fáze zpracování dotazu. Ač je tato třída obsáhlá, jejím úkolem je pouze řízení komunikace mezi servery a

zasílání dat. Samotné zpracování ať už na mapovacím uzlu nebo uzlu redukce je spouštěno třídami *MapPhase* a *ReducePhase*.

4.2.2 Průběh mapovací fáze

Map fáze na uzlu začíná přijetím příkazu pro map zpracování bloků dat. Na počátku proběhne znovu přeložení dotazu. I když byl dotaz překládán již na master uzlu pro kontrolu dotazu, není z důvodu omezení přenášených dat zasílán v každé map-reduce úloze. Proto proběhne jeho znovu přeložení na mapovacím uzlu, které není časově náročné. Bylo zmíněno, že mapovací uzel vykonává dávku mapovacích úloh. Jednotlivé úlohy postupně zpracovávají dotaz po jednom datovém bloku. Nyní si popíšeme průběh jedné mapovací úlohy.

Prvním krokem je zjištění, zda úloha obsahuje agregační funkci. Toto je rozhodující pro průběh celého vykonávání dotazu. Pokud funkce není přítomna a dotaz obsahuje pouze projekci požadovaných atributů a případně i selekci není prováděna fáze redukce, která seskupuje záznamy dle klíče, respektive dle požadovaných atributů. Tím pádem odpadá i rozdělování průběžných dat do celkem *R* částí. Po ukončení zpracování tohoto typu dotazu nejsou data přeposílána na uzly redukce, ale rovnou na master uzel, kde jsou dále seskupena a připravena k prezentaci.

V případě použití agregační funkce zpracování probíha v těchto hlavních krocích:

1. **třídění**, které proběhně dle seskupujících atributů,
2. **vyhledání** intervalů záznamů každého seskupení v rámci setříděné tabulky,
3. **agregace** dat podle zvolené funkce,
4. **výstup** dat ve formě nové tabulky.

Aby zpracování dotazu proběhlo sekvenčním průchodem všemi záznamy, bylo jako první krok určeno třídění. Třídění bylo provedeno nejen podle seskupovacích atributů, ale navíc i podle R-té části (primitivní datový typ *int*), do které jsou výsledky map fáze rozdělovány, což je první atribut, dle kterého je porovnávání během třídění vyhodnoceno. Tímto se sice zvedne časová náročnost třídění, která však ve výsledku další zpracování zefektivní.

Během třídění se také zaznamenávají unikátní klíče ve formě hodnoty primitivního typu *int*, kde jeden klíč představuje množinu hodnot seskupujících atributů záznamu. Díky tomu pak poměrně efektivně zjístíme rozsah seskupených záznamů v rámci tabulky. Funkci pro výpočet agregace se pak předává reference na celou tabulku, tedy na instanci *MemTable* a rozsah pro každé seskupení. Zároveň funkce provádí selekci a do výsledku zahrnuje záznamy splňující podmínku definovanou v dotazu. Takto je v cyklu agregováno každé seskupení.

Během zpracování je pro každý zpracovaný blok vytvořena nová tabulka obsahující selektované záznamy s požadovanými atributy, která dále obsahuje indexovou a vsm strukturu. Navíc je v paměti uchováno i rozdělení záznamů podle R-částí, což pak umožní jednoduchý výběr rozsáhu záznamů pro konkrétní uzel redukce.

Při popisování mapovací fáze v teoretické části práce bylo zmíněno, že data nejsou vždy agregována už mapovací fázi. Agregace je v této fázi použita jen v případě, že funkce splňuje *komutativnost* a *asociativnost*. Pokud funkce tyto dvě podmínky nesplňuje, odpadá tak výše uvedené zpracování agregace. Místo toho se postupně prochází všechny záznamy z důvodu selekce. Pokud konkrétní záznam podmínku splňuje, je přidán mezi výstupní data.

Celkovým výstupem mapovací fáze jsou tedy mezivýsledky ve formě nových tabulek, jejichž počet se rovná počtu datových bloků zdrojové tabulky. Vzhledem k velkému množství vytvářených dat je během této fáze nutné maximálně kontrolovat alokovanou paměť a průběžně provádět *delete* nad nepotřebnými daty, aby tak nedošlo k vyčerpání operační paměti.

4.2.3 Zasílání průběžných dat

Každý mapovací uzel po ukončení přiřazené úlohy odesílá na master notifikační zprávu o ukončení mapovací úlohy. Jakmile master obdrží tyto notifikace z každého uzlu, je mapovací fáze celého dotazu ukončena. Pokud v rámci dotazu není prováděna agregace, není nutné rozesílat data na uzly redukce, a proto jsou mezivýsledky zasílány na mapovací uzel, kde jsou seskupeny do jedné tabulky (v několika blocích), a připraveny pro prezentaci dat.

V případě provádění agregace v dotazu, je na každý uzel redukce zaslána zpráva obsahující hodnotu *i*-té části, kdy $i <= R$, kterou konkrétní uzel zpracuje a adresy mapovacích uzlů s mezivýsledky mapovací fáze. Na tyto adresy se pak uzel redukce dotazuje o zaslání jeho přidělené *i*-té části.

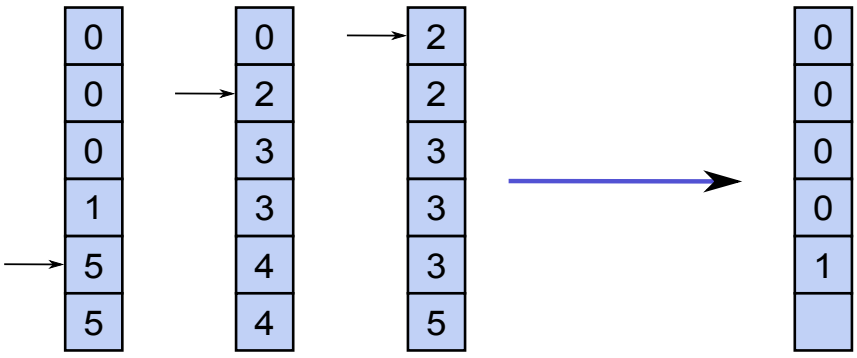
V této chvíli probíhá největší zatížení sítě neboť se má uskutečnit celkem $M \times R$ přenosů. Mírným ulehčením od zátěže sítě jsou situace, kdy uzel redukce je spuštěn na stejném uzlu jako mapovací uzel. Data toho mapovacího uzlu tedy nemusí být přenášena a jsou již připravena na konkrétním uzlu.

4.2.4 Průběh fáze redukce

Po přijetí všech požadovaných R-částí mezivýsledků (nových tabulek) je na uzlu redukce spuštěna fáze redukce. Zpracování dotazu se příliš neliší od mapovací fáze, neboť i zde zpracováváme jednu tabulku, respektive několik datových bloků tabulky, a nad seskupenými daty provedeme agregační funkci. Pokud bychom měli uvést jednotlivé kroky, jako při popisu mapovací fáze, uvedli bychom ty samé s tím rozdílem, že třídění je prováděno externí.

Hlavním důvodem pro externí třídení známé taky jako *merge sort* je objem dat. Při velkém objemu dat nemusí stačit operační paměť. Jednoduchý třídící algoritmus pro třídění objemných dat pracuje na principu rozdělení vstupních dat do bloků. Nejprve se každý blok setřídí zvlášť podle stejných parametrů třídění jako pro externí třídění, a dále se načte jen část záznamů z každého bloku a porovnávají se hodnoty podle ukazatele. Ukazatel v každé části z počátku ukazuje na první záznam a na další se posune až tehdy, pokud je jeho aktuální záznam vyhodnocen jako nejmenší či největší. Takto se projde

celou načtenou částí a případně se načte další část bloku. Postup externího třídění je pro srozumitelnost uveden na obrázku 3.



Obrázek 4: Ukázka principu externího třídění. Z každého bloku jsou porovnávány hodnoty určené ukazatelem. V tom bloku, ze kterého je při porovnání hodnota vybrána, se posune ukazatel o index dál.

V našem případě máme tu výhodu, že data již máme do bloků rozdělena a navíc i setříděna díky třídění v mapovací fázi. Stačí tedy provést jen popsaný algoritmus třídění. Z každým blokem lze pracovat jako s tabulkou pomocí třídy MemTable a práce je tak při třídění usnadněna. Výsledkem je nová setříděná tabulka, která je ukládaná v blocích na disk.

Dále je průběh zpracování totožný s mapovací fází, až na to, že není třeba zaznamenávat rozdělení dat v rámci některých částí (v mapovací fázi R-části) a zpracování se nevětví na zpracování s agregací nebo bez. Tady je zkrátka agregace vykonána vždy. Výstupem je tabulka určená pro zaslání na master uzel. Po přijetí výsledků z každého uzelu redukce jsou seskupena připravena pro vyžádaní klientem a map-reduce úloha je ukončena.

4.3 Implementace překladu dotazu

Zvolený dotaz na data je potřeba přeložit, aby byly během zpracování dat k dispozici všechny potřebné informace o požadovaném dotazu. K tomuto účelu je navrhnuta struktura, která reprezentuje dotaz parsovaný z textového řetezce. Tato struktura je popsána třídou *SQLContent*, viz dříve zmiňovaný obrázek 2. Textový řetězec SQL je postupně procházen pro vyhledání jednotlivých klauzulí a jejich obsahu a následně proběhne kontrola na správnost použití klauzulí a názvů atributů.

Za zmínku stojí implementace klauzule WHERE, která je uložena do stromové struktury pomocí tří typů *struct*, a to *ConditionTree*, *ConditionNode* a *ConditionUnit*. ConditionTree představuje celou klauzuli a obsahuje referenci na kořenový uzel tromu, instance ConditionNode. Tato třída obsahuje buď samotný výraz podmínky - ConditionUnit a stává se tak listem stromu, nebo obsahuje referenci na následující dvě instance ConditionNode a k nim typ konjunkce *LogicConEnum*, tedy AND nebo OR. Při vyhodnocení podmínky konkrétního záznamu je nejprve volána funkce *void evaluateCondition*, která

v případě existence stromu podmínek volá funkci *void evaluateNode*, jinak vyhodnotí samotný výraz (funkce *void evaluateExpression*, není zde uvedena). V této funkci se od kořene stromu iterativně prochází uzly stromu a v případě, že uzel již následující uzly neobsahuje (hodnota enumu pro konjunkci je null), pak se jedná o list stromu a volá funkci pro vyhodnocení výrazu, viz výpisy kódu 1 a 2.

```
void evaluateCondition(MemTable * table, ConditionTree & cond, long long & row, bool & result) {
    if (cond.root.conjuction == null) {
        result = evaluateExpression(table, cond.root, row);
    } else {
        result = evaluateNode(table, cond.root, row);
    }
}
```

Výpis 2: Funkce pro vyhodnocení podmínek klauzule WHERE jednoho záznamu

```
bool evaluateNode(MemTable * table, ConditionNode & node, long long & row) {
    if (node.conjuction == null) {
        return evaluateExpression(table, node, row);
    } else {
        bool * results = new bool[node.nextNodes.size()];
        for(unsigned int i = 0; i < node.nextNodes.size(); i++) {
            results[i] = (evaluateNode(table, *node.nextNodes[i], row));
            if (results[i]) {
                if (node.conjuction == OR) {
                    return true;
                }
            } else {
                if (node.conjuction == AND) {
                    return false;
                }
            }
        }
        if (node.conjuction == AND) { // AND
            for(unsigned int i = 0; i < node.nextNodes.size(); i++) {
                if (results[i] == false) {
                    return false;
                }
            }
            return true;
        } else { // OR
            for(unsigned int i = 0; i < node.nextNodes.size(); i++) {
                if (results[i] == true) {
                    return true;
                }
            }
            return false;
        }
    }
}
```

Výpis 3: Funkce pro vyhodnocení uzlu stromu podmínek klauzule WHERE

Dále je za pomocí virtuálních metod simulována konstukce "rozhraní"(interface), tedy vytvoření třídy IAggregationFunction obsahující pouze deklarace virtuálních metod, které musí implementovat třídy dědící z této. Přináší to výhodu jednotného tvaru tříd představující samotné agregační funkce. Během zpracování dat se pak volá metoda *compute* třídy IAggregationFunction, která provede agregaci nezávisle na typu funkce. Dále to přináší možnost doimplementace nových funkcí do systému dle daného předpisu.

4.4 Fyzická implementace datových stuktur

Fyzická implementace dat je samostatná problematika databázových systémů, jejíž vhodný návrh je základem pro efektivní dotazování [20]. Jak řeší fyzickou implementaci výše popisované nebo jiné NoSQL databázové systémy je dáno datovým modelem, zda se jedná o data ve formě tabulek, dokumentů, grafová data, či jinou formu dat. Pro tuto implementaci je zvolena forma tabulek. Ve stručnosti lze říct, že záznamy jsou v tabulce uloženy buď náhodně (heap tabulky), kdy nejsou v tabulce uspořádány podle jakéhokoliv kritéria, nebo jsou shlukovány podle požadovaných klíčů (shlukované záznamy, hash tabulky). V prvním případě jsou záznamy fyzicky uspořádány tak, jak byly postupně do tabulky vkládány, v druhém případě jsou záznamy fyzicky uspořádány ve vhodné datové struktuře (B-strom), která zajistí uspořádání dle požadovaného klíče. V této práci neimplementujeme žádné uspořádání záznamů, do datového bloku jsou postupně ukládány na konec. Tato implementace je vhodná z důvodu její jednoduchosti, rovněž je pro následující testovací účely dostatečná.

Datový soubor obsahuje samotná data z importovaného souboru. Soubor je jeden velký řetězec bajtů, kdy hodnoty záznamů jsou uloženy jako datové typy (serializované datové typy int, double a char *, unsigned short či unsigned char). Před každým záznamem je navíc jedinečné ID záznamu, značící pořadí záznamu v datovém souboru. Vzhledem k volené maximální velikosti datového bloku je těchto souborů vytvořen patřičný počet pro zahrnutí všech dat. Pokud jsou data takto rozdělena do více souborů, jejich velikost se může mírně lišit. Je to z důvodu kontroly velikosti importovaného záznamu – pokud vložením záznamu bude přesáhnuta maximální povolená velikost bloku, stávající blok dat se uloží a vytvoří se nový, do kterého se pak záznam vloží. Nenastane tedy situace, kdy záznam nebude v bloku kompletní. Součet počtů všech výsledných datových bloků na každém uzlu nám pak udává hodnotu M, viz. popis modelu v teoretické části.

V datovém souboru se mohou vyskytovat záznamy s atributy typu VARCHAR, kdy mají různou délku. Díky tomu nelze ze schématu určit délku záznamu v datovém souboru a nelze tedy využít pro přesun na následující záznamy jednotnou hodnotu počtu bytů. Byla proto zaveden **Indexový soubor** - indexace každého záznamu.Index záznamu je počátek pozice první hodnoty záznamu, což je výše zmiňované ID. Soubor je pole hodnot typu long long. První hodnota značí indexovaný atribut, následují hodnoty indexů, jejichž počet se rovná počtu záznamů, a poslední hodnota je celková velikost souboru (pomocná kontrolní hodnota).

První hodnota značící indexovaný atribut je zde pro to, abychom mohli zaindexovat nejen pozice záznamů v souboru, ale také jejich vybrané atributy. Pokud má datový soubor velké množství atributů, jsou některé operace, zejména třídění, výrazně zpomaleny

"dopočítáváním"pozice hodnoty od počátku záznamu. Proto je indexace jiných atributů umožněna a při vytváření indexů atributu jsou indexy zapisovány na konec indexového souboru dle stejného pravidla jako u indexace záznamů, až na to, že hodnoty těchto indexů nejsou pozice v rámci datového souboru, ale offset od počátku záznamu.

Počet indexových souborů je roven počtu datových souborů. Obsahují tedy jen indexy záznamů v odpovídajících datových souborech. Vzhledem k tomu, že je možné kdykoliv indexovat atributy, není maximální velikost bloku indexů omezena.

Znalost délek řetězců hodnot atributů jednotlivých záznamů je důležitá pro nalezení konkrétní hodnoty v datovém souboru. Toto řeší struktura **Varchar size map** (dále jen zkráceně VSM), která popisuje délky každého řetězce každého záznamu. Bylo uvedeno, že datové soubory uchovávají v případě datového typu VARCHAR i nulový znak, tedy znak '\0', díky čemuž lze zjistit délku řetezce např. pomocí funkce *strlen(const char * c)*. Při zpracování několika milionů záznamů se však použití vypočtu délky řetezce může stát časově náročné. Z toho důvodu bylo rozhodnuto o použití VSM. To však sebou přináší další množsví dat, které nemusí být zanedbatelné. Proto v následujících testech je provedeno ověření, zda je použití VSM vhodné i přes vznik dalších dat, nebo bude dostačující využít výpočet délky řetězce během zpracování.

VSM soubor je podobný indexovému souboru, ale hodnoty jsou uloženy jako datový typ int. První hodnota značí atribut a dále následují hodnoty, jejichž počet je roven počtu záznamů. V případě více atributů jsou stejným pravidlem hodnoty přidávány na konec souboru. I tyto soubory jsou rozděleny dle datových souborů a nejsou jinak omezeny na velikost.

Řádkový soubor obsahuje počty záznamů v jednotlivých blocích dat. Pro každý blok dat obsahuje dvě hodnoty datového typu long long. První je počátek bloku dat vzhledem k celkovému počtu záznamů (je to v podstatě pořadí prvního záznamu v bloku) a druhá je počet záznamů v bloku.

Schéma tabulky je z textového souboru převedeno do binárního tvaru. Každý atribut je zapsán jedním párem hodnot. První je řetězec pevné velikosti 128 Bytů a druhá hodnota je číslo datového typu int, která značí datový typ atributu dle enumu *DataType*.

4.4.1 Třídy MemTable a CharBuffer

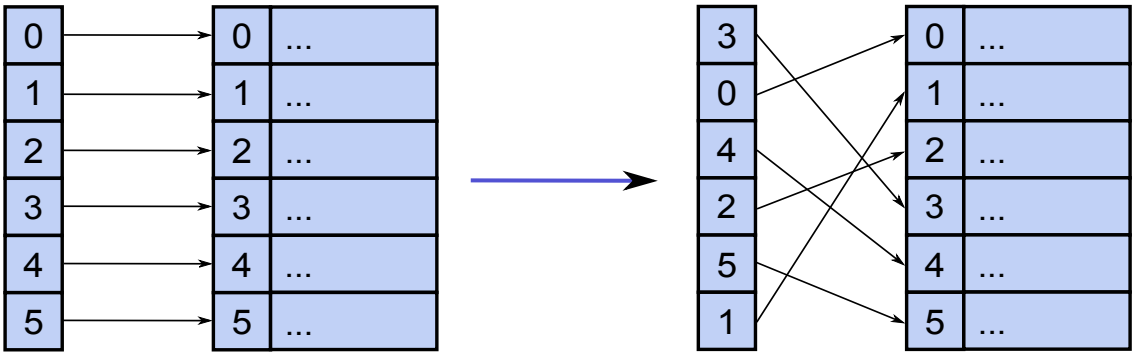
Výše popsané struktury popisují jednu tabulku. Tabulka, respektive jeden její datový blok včetně popisujících dat jako indexy či VSM, jsou v průběhu každé fáze vykonávání map-reduce úlohy načteny v paměti. Aby se se všemi těmito strukturami pracovalo jednoduše jako s tabulkou, případně vytvářela se nová tabulka, byly implementovány třídy *MemTable* a *CharBuffer*.

Třída MemTable provádí načtení všech struktur popsaných v předchozí podkapitole, tedy kompletní blok tabulky nebo dokonce jen jeho část. V případě, kdy blok tabulky obsahuje 1 000 000 záznamů, je možné načíst jen požadovaný souvislý interval záznamů, např. prvních 10 000 záznamů apod. Při zpracování v mapovací fázi nebo ve fázi redukce je blok načten kompletní, ale možnost načíst jen část bloku je využito v případě provádění externího třídění (známé taky jako *merge sort*, podrobněji v podkapitole o fázi redukce).

K datům se pak přistupuje především pomocí následujích funkcí:

- *int * getIntValue(const long long & row, const int & col)* - vrací ukazatel na hodnotu typu INTEGER požadovaného řádku(záznamu) a sloupce(atributu),
- *double * getDoubleValue(const long long & row, const int & col)* - vrací ukazatel na hodnotu typu DOUBLE požadovaného řádku(záznamu) a sloupce(atributu),
- *char * getVarcharValue(const long long & row, const int & col)* - vrací ukazatel na hodnotu typu VARCHAR požadovaného řádku(záznamu) a sloupce(atributu),
- *char * getBuffer(long long & row)* - vrací ukazatel na počátek záznamu v bufferu. Využívá se zejména v případě potřeby kopírovat celý záznam.

Výhodou použití třídy MemTable je také implementace funkce pro třídění dle jednoho či více atributů různých typů. Z důvodu jednoduchosti a efektivity se při třídení netřídí samotné záznamy v bufferu, ale jen pole ukazatelů na tyto záznamy. Pole ukazatelů je vytvořeno při načtení bloku tabulky do paměti za pomocí indexů. V praxi to pak znamená, že po setřídení při zavolání metody pro získání konkrétní hodnoty např. prvního záznamu je vrácena hodnota záznamu, který je fyzicky uložen v datech jako stý záznam. Toto je jeden z důvodů proč bylo na počátek každého záznamu při importu přidáno jeho ID - pořadí záznamu v datovém bloku. Jeho originální pořadí je pak využíváno při získávání indexů záznamů či atributů nebo délek řetezců, které jsou načteny v poli. Implementace s využitím ukazatelů na záznamy do bufferu před a po třídení je ilustrováno na obrázku č. 4.



Obrázek 5: Ukázka dat s využitím ukazatelů na data před a po setřídění.

Samotné třídení není provedeno vlastní implementací, ale je využito standartní C++ funkce *void qsort_s(void * base, size_t num, size_t width, int (__cdecl * compare)(void *, const void *, const void *), void * context)* implementují třídící algoritmus *Quick sort*. K použití této funkce bylo nutné implementovat vlastní *compare* funkci, tedy funkci, která provádí porovnaní dvou prvků tříděného pole.

Dále je implementována třída CharBuffer, což je třída využívána pro zjednodušení práce s nově vytvářenými řetězci znaků, zejména při importu dat, vytváření mezivýsledků pro fázi redukce a také i při externím třídění. Výhoda použití spočívá v automatickém

odkládání dat na disk v případě překročení zvolené maximální velikosti, kterou má udržovat v paměti. Třídě lze také nastavit příznak blokující automatické odkládáním, které je pak nutné řešit manuálně. Při používaní této třídy jsou využity nejčastěji tyto funkce:

- *void add(char * data, long long cnt)* - vloží do bufferu zvolený počet bytů ze zdrojového pole),
- *char * get(long long startPos, unsigned int cnt)* - vrací ukazatel do bufferu na zvolenou pozici
- *char * getVarcharValue(const long long & row, const int & col)* - vrací ukazatel na hodnotu typu VARCHAR požadovaného řádku(záznamu) a sloupce(atributu),
- *void storeBuffer()* - uloží buffer do souboru a alokuje místo pro nový blok dat, které při dalším zavolání této funkce uloží do nového souboru.

5 Experimenty a porovnání implementace

V této kapitole jsou popsány testy implementovaného systému. V první části jsou testy zaměřeny na optimální nastavení systému a v druhé části jsou testy zaměřeny na porovnání této implementace MapReduce s jiným vybraným databázovým systémem. Testy jsou vždy spouštěny nad generovanými daty a jejich různým objemem.

5.1 Testovací prostředí

Pro simulaci clusteru uzlů pro zpracování map-reduce úloh byly využity školní servery *DBEDU* a *DBSYS* ve spojení s vlastním PC připojeným do školní sítě pomocí VPN. Na těchto serverech je v následujících testech podle potřeb konkrétního testu spuštěno několik instancí (dále jako uzel) implementované aplikace, kdy každá pak naslouchá na jiném portu. U každého uzlu musí být také nastavena tabulka známých serverů a jejich portů. V implementaci tedy není řešen způsob "učení se"uzlu o znalosti svého okolí (ostatních uzlů) pomocí zasílání broadcast zpráv či jiným způsobem. Vzhledem k ovlivnění měřených časů nízkou propustností sítě je rozložení uzlů směrováno především na servery *DBEDU* a *DBSYS* a vlastní PC je využito pouze pro spuštění klienta, případně Master uzlu. Největší zatížení sítě nastává při přeposílání dat mezi uzly, proto budou spouštěny na školních serverech, které jsou propojeny 100 Mbit switchem. Ze stejného důvodu není měřen čas včetně zasílání výsledků klientovi, ale jen po ukončení zpracování.

5.2 Testovací data a dotazy

Data jsou generována open-source nástrojem Spawner Data Generator [39]. Nástroj je pro naše testy plně dostačující, neboť obsahuje generování, pro nás podstatných, datových typů. Pro testování implementace MapReduce není nutné generovat specifická data, jen je důležité v případě využití replikace zajistit kopie dat na všech uzlech zapojených do zpracování dotazu. Počet generovaných záznamů pro konkrétní testy se liší podle potřeby. Jsou vygenerovány celkem dvě tabulky, nad kterými je spuštěn každý test. Jsou generovány tak, abychom otestovali implementaci na různými variacemi datových typů. Schémata tabulek jsou popsána následujícími tabulkami a ukázky vygenerovaných dat se nachází v příloze. Atributy pro každé schéma byly vybrány tak, aby odpovídaly běžně využívaným tabulkám v praxi, viz tabulka 2 a 3.

V tabulce 4 jsou vypsány testovací dotazy prováděné nad tabulkou Zamestnanec a testovací dotazy pro tabulky Pocasi jsou vypsány v tabulce 5. Dotazy byly vybrány tak, aby pokryly implementovanou podmnožinu SQL a testovanou funkcionalitu. Typově se dotazy nad ruznými tabulkami neliší. Jedná se o dotazy na konkrétní záznam podle klíče, dotazy na podmnožinu záznamů specifikované podmínkou a agregované dotazy.

5.3 Testování optimálního nastavení

Pro zjištění optimálního nastavení systému jsou provedený celkem čtyři testy. První dva testy se týkají využití indexového souboru a VSM souboru při zpracování dotazu. Cílem

ZAMESTNANEC		
pořadí	**název atributu**	**datový typ**
0	id	integer
1	jmeno	varchar
2	prijmeni	varchar
3	datum_narozeni	datetime
4	email	varchar
5	telefon	varchar
6	ulice	varchar
7	cislo_popisne	varchar
8	mesto	varchar
9	stat	varchar
10	psc	varchar
11	kancelar	varchar
12	klapka	integer
13	oddeleni	varchar
14	funkce	varchar
15	plat	integer
16	nastup	datetime
17	zmena_udaju	datetime

Tabulka 2: Atributy tabulky Zamestnanec představující seznam zaměstnanců.

POCASI		
pořadí	**název atributu**	**datový typ**
0	id	integer
1	kod_stanice	varchar
2	cas	datetime
3	teplota_zem	double
4	teplota_metr	double
5	teplota_nadzem	double
6	tlak	double
8	vitr	double
9	vlhkost	integer
10	dohled	integer
11	svetelnost	integer

Tabulka 3: Atributy tabulky Pocasi představující seznam meteorologická měření.

id	**dotaz**
Q0	SELECT * FROM Zamestnanec WHERE id = 4444444;
Q1	SELECT jmeno, prijmeni, datum_narozeni FROM Zamestnanec WHERE mesto = 'Minneapolis';
Q2	SELECT jmeno, prijmeni, zmena_udaju FROM Zamestnanec WHERE zmena_udaju >= '2000-01-01 00:00:00' AND zmena_udaju <= '2005-01-01 00:00:00';
Q3	SELECT COUNT(*) FROM Zamestnanec;
Q4	SELECT MAX(plat) FROM Zamestnanec WHERE stat='Alaska';
Q5	SELECT stat, AVG(plat) FROM Zamestnanec GROUP BY stat;
Q6	SELECT stat, MIN(plat), MAX(plat) FROM Zamestnanec WHERE oddeleni='MNG' GROUP BY stat;

Tabulka 4: Testované SQL dotazy nad tabulkou Zamestnanec.

těchto testů je ověřit, zda jejich využití přinese větší efektivitu i přes jejich nezanedbatelný objem, který je nutné vždy brát v potaz vzhledem k distribuovanosti systému, respektive propustnosti datové sítě. Další test je proveden na vhodné nastavení velikosti vstupních datových bloků a poslední test se týká nastavení optimální hodnoty M a R.

id	dotaz
Q7	SELECT * FROM Pocasi WHERE id = 4444444;
Q8	SELECT id, kod_stanice, cas FROM Pocasi WHERE dohled = 50;
Q9	SELECT id, kod_stanice, svetelnost FROM Pocasi WHERE cas >= '2000-01-01 00:00:00' AND cas <= '2005-01-01 00:00:00';
Q10	SELECT COUNT(*) FROM Pocasi;
Q11	SELECT MAX(teplota_nadzem) FROM Pocasi WHERE kod_stanice='PRA';
Q12	SELECT kod_stanice, AVG(teplota_metr) FROM Pocasi GROUP BY kod_stanice;
Q13	SELECT kod_stanice, MIN(teplota_metr), MAX(teplota_metr) FROM Pocasi WHERE vlhkost=90 GROUP BY kod_ stanice;

Tabulka 5: Testované SQL dotazy nad tabulkou Pocasi.

V každém následujícím testu je využito plné replikace dat. Každá testovaná tabulka je v testech použita ve dvou verzích - tabulka s 5 mil. záznamy (verze A) a tabulka s 10 mil. záznamy (verze B). V tabulce 6 je uveden počet záznamů každého dotazu pro jednotlivé verze a v tabulkách 7 a 8 počet výsledků každého dotazu. V testech č. 1, 2 a 3 je využito dvou pracovních uzlů, kdy je každý z nich spuštěn školních serverech, master a klient je umístěn na PC. Rozložení je graficky znázorněno na obrázku. Pro test č. 4 je využito celkem pět pracovních uzlů, rozložení je blíže popsáno v samotném testu.

tabulka	verze	počet záznamů	velikost csv [MB]	velikost d. s. [MB]
Zamestnanec	A	5 000 000	998	1 178
	B	10 000 000	1 997	2 357
Pocasi	A	5 000 000	351	381
	B	10 000 000	704	1 106

Tabulka 6: Velikosti generovaných verzí tabulek v csv formátu a velikost datových struktur databázového systému.

verze	Q0	Q1	Q2	Q3	Q4	Q5	Q6
A	1	4 167	1 924 349	1	1	51	51
B	1	9 538	3 847 525	1	1	51	51

Tabulka 7: Počet záznamů výsledků každého dotazu pro obě verze tabulky *Zamestnanec*.

verze	Q7	Q8	Q9	Q10	Q11	Q12	Q13
A	1	41 722	1 576 723	1	1	19	19
B	1	82 661	3 155 796	1	1	19	19

Tabulka 8: Počet záznamů výsledků každého dotazu pro obě verze tabulky *Pocasi*.

5.3.1 Test č. 1 - indexové soubory

Bylo popsáno, že indexy jsou využity pro indexaci záznamů (pozice záznamu v datovém bloku) a také pro indexaci hodnot vybraných atributů (pozice hodnoty od počátku atributu). Při čtení konkrétní hodnoty atributu v záznamu je nutné znát jeho pozici, kterou získáme dopočtem délky v bytech předchozích hodnot. Během iterativní implementace systému a jeho ladění byly zjištěny nemalé výkonnostní rozdíly nejen při třídění záznamů podle atributů s nízkou pořadovou hodnotou oproti atributům s vysokou pořadovou hodnotou (např. třídění dle druhého atributu oproti třídění dle desátého atributu). Cílem testu č. 1 je přesné porovnaní výkonu systému při využití či nevyužití indexace atributů. V testu je využito VSM. V tabulkách 9 a 10 jsou zobrazeny časové výsledky zpracování testovaných dotazů, v tabulce 11 pak přehledněji časový rozdíl zpracování s využitím indexů.

verze	A		B	
dotaz	čas bez indexace [s]	čas s indexací [s]	čas bez indexace [s]	čas s indexací [s]
Q0	1,186	1,179	2,189	2,205
Q1	1,812	1,741	3,776	3,088
Q2	16,798	10,167	32,469	20,073
Q3	1,614	1,655	2,613	2,683
Q4	2,461	2,533	4,463	4,588
Q5	30,557	19,322	75,661	45,461
Q6	27,428	14,554	57,827	30,048

Tabulka 9: Čas zpracování dotazů nad oběma verzemi tabulky *Zamestnanec* bez indexace a s indexací atributů *mesto*, *stat*, *oddeleni* a *zmena_udaju* a využitím VSM metadat.

verze	A		B	
dotaz	čas bez indexace [s]	čas s indexací [s]	čas bez indexace [s]	čas s indexací [s]
Q7	0,830	0,860	1,52	1,642
Q8	1,382	1,424	2,644	2,692
Q9	3,784	3,872	7,541	7,602
Q10	1,040	1,050	1,752	1,762
Q11	1,320	1,391	2,250	2,377
Q12	12,931	13,032	26,904	30,218
Q13	7,522	7,658	15,034	15,185

Tabulka 10: Čas zpracování dotazů nad oběma verzemi tabulky *Pocasi* bez indexace a s indexací atributů *kod_stanice* a *teplota_metr* a využitím VSM metadat.

Zavedením indexů se zvedl objem dat o nezanedbatelné množství, nicméně provedení dotazů nad tabulkou *Zamestnanec* bylo podstatně rychlejší než bez indexace. Již nad daty o 5 mil. záznamech získáme téměř o polovinu rychlejší zpracování u některých dotazů. Proto je zavedení vhodných indexů velmi vhodné i přes jejich velikost, obzvlášť u zpracování

tabulka	verze	jedn.	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
Zamestnanec	A	[s]	0,007	0,071	6,631	-0,041	-0,072	11,235	12,874
		[%]	0,59	3,92	39,47	-2,54	-2,93	36,77	46,94
	B	[s]	-0,016	0,688	12,396	-0,070	-0,125	30,200	27,779
		[%]	-0,73	18,22	38,18	-2,68	-2,80	39,81	48,04
			Q7	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**
Pocasi	A	[s]	-0,030	-0,042	-0,088	-0,010	-0,071	-0,101	-0,136
		[%]	-3,61	-3,04	-2,33	-0,96	-5,38	-0,78	-1,81
	B	[s]	-0,122	-0,048	-0,061	-0,010	-0,127	-3,314	-0,151
		[%]	-8,03	-1,82	-0,81	-0,57	-5,64	-12,32	-1

Tabulka 11: Rozdíl časů zpracování dotazů nad tabulkou *Zamestnanec* s využitím indexů nad atributy *mesto*, *stat*, *oddeleni* a *zmena_udaju* a nad tabulkou*Pocasi* bez využití a s využitím VSM a indexů nad atributy *kod_stanice* a *teplota_metr*

mnohem objemnějších dat. V dotazech však nastaly výjimky, kde zavedením indexů nenastalo žádné zlepšení či dokonce nastalo mírné zpomalení (max. desetiny vteřin). Je to z toho důvodu, že v dotazech nejsou vůbec použity indexované atributy a mírné zpomalení je způsobeno načítáním nevyužitých indexů při načítání datového bloku.

Naopak u tabulky *Pocasi* ve většině případů nastalo mírné zpomalení. Vysvětlením pro tuto situaci je celková velikost indexů v porovnání s celkovou velikostí dat tabulky. Vzhledem k menšímu počtu atributů a jejich datových typů má při stejném počtu záznamů tabulka *Pocasi* mnohem menší velikost než tabulka *Zamestnanec*. Velikost indexů pro jeden atribut však zůstává stejná. Pak práce s indexy během zpracování dotazů nepřináší zvýšení efektivity.

Pro další testy (kromě testu 2) jsou u tabulek ponechány jen ty indexy atributů, které svým zavedením výrazně zlepšily zpracování dotazu. U tabulky *Zamestnanec* to jsou atributy *stat* a *zmena_udaju*, u tabulky *Pocasi* pak nebyly ponechány žádné atributy.

5.3.2 Test č. 2 - VSM soubory

V případech, kdy ve schématu tabulky existuje větší množství atributů typu Varchar, může velikost VSM dat narůst poměrně vysoko. Při výpočtu pozice konkrétní hodnoty v záznamu je využito VSM hodnot. Vzhledem k možnému vysokému objemu těchto metadat je cílem testu ověření, nakolik je vhodné metada při zpracování využívat, nebo je efektivnější zjišťovat délku Varchar hodnoty pomocí standartní funkce *strlen(const char *)*. Test je úzce spjat s předchozím testem č. 1, neboť v případě neindexovaných atributů je nutné pozici hodnoty v záznamu dopočítat, což je zcela běžná situaci, protože indexace zpravidla není zavedena nad větším množstvím atributů. Test je proveden před i po zavedení indexace. Stejně jako u předchozího testu, v tabulkách 12 a 13 jsou zobrazeny časové výsledky zpracování dotazů bez VSM a následně v tabulkách 14 a 15 přehledněji časový rozdíl zpracování s využitím VSM. Pro porovnání jsou využity naměřené hodnoty z předchozího testu, kdy VSM byly při zpracování využity.

verze	A		B	
dotaz	čas bez indexace [s]	čas s indexací [s]	čas bez indexace [s]	čas s indexací [s]
Q0	1,127	1,187	2,048	2,417
Q1	1,926	2,003	3,731	3,681
Q2	12,039	13,373	24,038	26,238
Q3	1,408	1,632	2,408	3,005
Q4	2,971	2,987	5,661	5,519
Q5	33,988	21,879	82,185	49,032
Q6	30,995	14,801	59,220	31,263

Tabulka 12: Čas zpracování dotazů bez využití VSM nad oběma verzemi tabulky *Zamestnanec* bez indexace a s indexací atributů *mesto*, *stat*, *oddeleni* a *zmena_udaju*.

verze	A		B	
dotaz	čas bez indexace [s]	čas s indexací [s]	čas bez indexace [s]	čas s indexací [s]
Q7	0,972	0,952	1,474	1,540
Q8	1,412	1,420	2,676	2,71
Q9	3,816	3,820	7,542	7,55
Q10	1,071	1,032	1,682	1,732
Q11	1,424	1,331	2,21	2,276
Q12	12,08	12,326	26,384	26,554
Q13	7,682	7,892	15,174	15,304

Tabulka 13: Čas zpracování dotazů bez využití VSM nad oběma verzemi tabulky *Pocasi* bez indexace a s indexací atributů *kod_stanice* a *teplota_metr*.

verze	indexy	jedn.	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
A	ne	[s]	-0,059	0,114	-4,759	-0,206	0,510	3,431	3,567
		[%]	-5,24	5,92	-39,53	-14,63	17,17	10,09	11,51
	ano	[s]	0,008	0,262	3,206	-0,023	0,454	2,557	0,247
		[%]	0,67	13,08	23,97	-1,41	15,20	11,69	1,67
B	ne	[s]	-0,141	-0,045	-8,431	-0,205	1,198	6,524	1,393
		[%]	-6,88	-1,21	-35,07	-8,92	21,16	7,94	2,35
	ano	[s]	0,212	0,593	6,165	0,322	0,931	3,571	1,215
		[%]	8,77	16,11	23,5	10,72	16,87	7,28	3,89

Tabulka 14: Rozdíl časů zpracování zpracování dotazů nad tabulkou *Zamestnanec* bez využití a s využitím VSM a indexů nad atributy *mesto*, *stat*, *oddeleni* a *zmena_udaju*.

Výsledky ukazují, že využití VSM u tabulky *Zamestnanec* přinese určité zrychlení jak u využití indexů tak bez indexů. Nevýhodou těchto dat je jejich velikost, která může tvořit i čtvrtinu celkové velikost tabulky s větším množstvím varchar atributů. Časový přínos není pak není příliš vysoký, obzvlášť u indexovaných dat. Proto nebudou VSM data v dalších testech u této tabulky využita.

verze	indexy	jedn.	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**
A	ne	[s]	-0,142	-0,030	-0,032	-0,031	-0,104	0,851	-0,160
		[%]	-17,11	-2,17	-0,85	-2,98	-7,88	6,58	-2,13
	ano	[s]	-0,092	0,004	0,052	0,018	0,060	0,706	-0,234
		[%]	-10,7	0,28	1,34	1,71	4,31	5,42	-3,06
B	ne	[s]	0,046	-0,032	-0,001	0,070	0,040	0,520	-0,140
		[%]	3,03	-1,21	-0,01	4	1,78	1,93	-0,93
	ano	[s]	0,102	-0,018	0,052	0,030	0,101	3,664	-0,119
		[%]	6,21	-0,67	0,68	1,7	4,25	12,13	-0,78

Tabulka 15: Rozdíl časů zpracování zpracování dotazů nad tabulkou *Pocasi* bez využití a s využitím VSM a indexů nad atributy *kod_stanice* a *teplota_metr*.

Podobný výsledek nastal i tabulky *Pocasi*, kdy je zrychlení velmi nízké. Tabulka obsahuje pouze jeden atribut typu VARCHAR, proto ani není očekávána výhoda při použití VSM a v dalších testech také nebudou využita.

5.3.3 Test č. 3 – velikost vstupních dat

Základem efektivity databázových systémů je zajištění co nejmenšího počtu přístupu na disk. Snaha je mít načteno co nejvíc dat v operační paměti z důvodu rychlejšího přístupu do paměti oproti přístupu k datům na disk. V této implementaci jsou data rozdělována při importu do datových bloků o zvolené velikosti, kdy při zpracování dotazu je vždy v operační paměti načten právě jeden blok. Je proto vhodné zvolit co největší možnou velikost datového bloku. Cílem testu je ověření, zda opravdu je v této implementaci nejvýhodnější použití co největšího možného datového bloku, nebo může nastat vzhledem ke zpracování velkého objemu vstupních dat situace, kdy je výhodnější mít vytvořeny menší bloky.

Testované velikosti bloků jsou 64 MB, 128 MB, 256 MB a 512 MB.

		čas zpracování dotazu [s]						
verze	velikost [MB]	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
A	64	1,097	2,063	13,505	1,902	3,471	27,580	15,473
	128	1,070	2,074	13,101	1,748	3,461	26,865	16,711
	256	1,094	1,952	13,156	1,707	3,439	21,833	17,804
	512	1,163	1,957	13,085	1,783	3,342	24,182	18,058
B	64	2,291	3,773	24,580	3,511	6,515	61,789	27,230
	128	2,515	3,728	26,009	3,291	6,429	53,150	30,414
	256	2,361	3,872	26,196	3,114	6,233	50,955	36,521
	512	2,357	3,878	26,232	3,038	6,240	49,271	37,436

Tabulka 16: Čas zpracování dotazů nad tabulkou *Zamestnanec* s datovými bloky o různé velikost.

		čas zpracování dotazu [s]						
verze	velikost [MB]	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**
A	64	0,930	1,490	2,922	0,964	1,372	12,678	10,766
	128	0,900	1,552	3,692	1,102	1,360	12,008	10,802
	256	0,860	1,550	3,646	1,090	1,340	12,384	11,182
	512	1,142	2,030	5,086	1,320	1,632	13,716	14,442
B	64	1,384	2,290	5,512	1,542	1,892	27,240	16,716
	128	1,312	2,310	5,444	1,530	1,822	27,676	16,954
	256	1,642	3,002	7,160	1,790	2,480	28,980	21,796
	512	1,602	2,970	7,124	1,762	2,202	33,240	22,620

Tabulka 17: Čas zpracování dotazů nad tabulkou *Pocasi* s datovými bloky o různé velikost.

V případě tabulky *Zamestnanec* Výsledky testu v tabulce 16 a 17 ukazují, že u dotazů bez agregační funkce větší velikost datového bloku nepřináší výrazně výkonnější zpracování. U dotazů s agregační funkcí už rozdíl není nezanedbatelný. Nejvíce se změna velikosti bloku projeví mezi blokem o velikostech 64 a 128 MB a mezi bloky o velikostech 128 a 256 MB. Rozdíl mezi 256 a 512 MB velkým blokem je nízký, v některých případech dokonce nastalo mírné zhoršení výkonu zpracování. V těchto případech je pravděpodobně z důvodu vyšší náročnosti na třídění(více záznamů) během zpracování dotazů s agregační funkcí. Proto využijeme pro další testování bloky o velikostech 256 MB. U tabulky *Pocasi* je zhoršení výsledků s velikostí bloku větší nebo rovné 256 MB výraznější, proto u této tabulky pro další testování použijeme bloky o velikostech 128 MB.

5.3.4 Test č. 4 – nastavení hodnoty M a R

Při zpracování dat více uzly vhodné práci rovnoměrně rozdělit. Volba hodnoty M, tedy počet mapovacích uzlů, není uživatelsky volitelná a závisí na počtu uzlů. Podle teorie zpracování popsané v předchzích kapitolách, je M hodnota rovna počtu vstupních bloků ne na jednom uzlu ale celkově napříč clusterem. V implementaci této práce je návrh takový, že hodnota M je menší nebo rovna počtu uzlů a to z toho důvodů, že v práci nepředpokládáme uchování vstupních dat na distribuovaném souborovém systému. V případě využití plné replikace obsahují uzly stejné kopie všech datových bloků a při zpracování každý uzel zpracuje patřičný vstupní blok. Pokud plná replikace není využita, každý uzel musí zpracovat všechny své datové bloky, bez ohledu na jejich počet. Hodnota R je není omezena počtem uzlů. Cílem testu je ověřit efektivitu zpracování dotazu vzhledem k počtu uzlů redukce. V testu je využito celkem pěti uzlů, z nichž tři jsou umístěny na serveru DBEDU a dva na DBSYS. V tabulkách 18 a 19 jsou uvedeny časy zpracování dotazů s různým využitím mapovacích a redukčních uzlů pro tabulku Zamestnanec, pro tabulku Pocasi jsou časy v tabulkách 20 a 21.

Výsledek testu dopadl u obou tabulek podle očekávání, a sice čím větší paralelizace, tím rychlejší zpracování, viz. příloha. U dotazů, kde není využita agregační funkce a tedy není spuštěna fáze redukce, nezáleží na hodnotě R. Zároveň je otestován i případ, kdy

	čas zpracování dotazu [s]				
M	1	1	4	4	4
R	1	5	1	5	8
Q0	1,814	1,733	1, 091	1,122	1,102
Q1	3,306	3,297	1,985	1,964	1,971
Q2	11,434	11,465	6,542	6,517	6,608
Q3	2,294	2,403	1,382	1,335	2,031
Q4	4,989	5,282	1,705	1,999	2,264
Q5	35,934	32,53	18,087	12,657	12,279
Q6	27,545	27,386	8,584	8,737	9,034

Tabulka 18: Čas zpracování dotazů nad tabulkou *Zamestnanec* verze A s využitím různého počtu wrokerů při nasazení aplikace na pěti uzlech.

	čas zpracování dotazu [s]				
M	1	1	5	5	5
R	1	5	1	5	8
Q0	3,394	3,380	1,244	1,232	1,271
Q1	6,488	6,490	2,088	2,121	2,180
Q2	22,738	22,778	6,646	6,651	6,679
Q3	4,060	4,383	1,577	1,864	2,131
Q4	9,610	9,918	3,100	3,414	3,631
Q5	88,546	73,527	52,046	26,058	31,588
Q6	55,159	54,972	16,787	17,042	17,261

Tabulka 19: Čas zpracování dotazů nad tabulkou *Zamestnanec* verze B s využitím různého počtu wrokerů při nasazení aplikace na pěti uzlech.

	čas zpracování dotazu [s]				
M	1	1	3	3	3
R	1	5	1	5	8
Q7	1,214	1,112	0,600	0,572	0,570
Q8	2,052	2,082	0,860	0,850	0,860
Q9	5,062	5,061	1,942	1,932	1,942
Q10	1,264	1,608	0,680	0,940	1,180
Q11	1,770	1,892	0,952	1,182	1,990
Q12	17,546	14,986	13,318	8,104	10,302
Q13	12,734	12,162	6,112	5,784	6,620

Tabulka 20: Čas zpracování dotazů nad tabulkou *Pocasi* verze A s využitím různého počtu wrokerů při nasazení aplikace na pěti uzlech.

hodnota R je větší než počet uzlů redukce. Tento případ je méně efektivní oproti hodnotě R

	čas zpracování dotazu [s]				
M	1	1	5	5	5
R	1	5	1	5	8
Q7	2,122	2,134	0,830	0,830	0,820
Q8	4,040	4,060	1,520	1,522	1,562
Q9	10,124	10,112	3,742	3,752	3,732
Q10	2,250	2,592	1,060	1,420	1,620
Q11	3,179	3,316	1,432	1,832	1,876
Q12	39,017	32,002	25,782	18,236	18,258
Q13	25,393	24,343	13,560	9,462	10,582

Tabulka 21: Čas zpracování dotazů nad tabulkou *Pocasi* verze B s využitím různého počtu wrokerů při nasazení aplikace na pěti uzlech.

rovné počtu uzlů redukce. Každý uzel sice zpracovává o něco menší mnoštví dat, nicméně správa zasílání příkazů na uzly a jejich spouštění zpracování zpomalí. V Dalších testech tedy maximálně využijeme možnost paralelizace a hodnotu M i R nastavíme na celkový počet uzlů.

5.4 Testování rozsáhlých dat a větší distribuce

V této části testování jsou spuštěny výše uvedené dotazy nad rozsáhlejšími daty. Pro každou tabulku byly tentokrát vygenerovány tři verze(A, B a C) kdy jednotlivé verze obsahují 50 mil., 75 mil. a 100 mil. záznamů. Jejich velikosti jsou uvedeny v tabulce 22. Pro takto rozsáhlá data bylo nutné zavést větší paralelizaci pro dosažení výsledků v rozumném čase. Celkem se v clusteru nachází 10 pracovních uzlu, respektive 10 uzlů a jeden uzel navíc pro klienta a master uzel. Rozložení uzlů v testovacím prostředí je uvedeno na obrázku a časy zpracování dotazů nad uvedenými daty jsou uvedeny v tabulce.

Pro test nad tabulkou *Zamestnanec* byl přidán dotaz *Q14*: *SELECT mesto, AVG(plat) from Zamestnanec GROUP BY mesto*.

tabulka	verze	počet záznamů [mil.]	velikost csv [MB]	velikost d. s. [MB]
Zamestnanec	A	50	10 025	12 561
	B	75	15 038	18 842
	C	100	19 968	25 020
Pocasi	A	50	3 563	5 598
	B	75	5 345	8 397
	C	100	7 127	11 196

Tabulka 22: Velikosti generovaných verzí tabulek pro test zpracování rozsáhlých dat v csv formátu a v datových strukturách systému.

	čas zpracování dotazu [s]							
	Q0	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q14**
počet výs.	1	47 437	23 063 993	1	1	51	51	1057
A	3,135	4,64	27,334	4,81	7,91	123,207	40,281	140,825
B	4,214	6,35	40,493	6,416	10,189	178,820	54,26	209,371
C	6,025	8,991	62,15	8,112	14,782	251,332	85,924	311,003

Tabulka 23: Čas zpracování dotazů nad tabulkou *Zamestnanec* s počtem záznamů 50 mil. (A), 75 mil. (B) a 100 mil. (C).

	čas zpracování dotazu [s]						
	Q7	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**
počet výs.	1	631 117	22 765 912	1	1	19	19
A	1,609	3,064	6,828	2,715	3,070	59,878	19,728
B	2,480	4,460	10,042	3,874	4,582	88,84	28,593
C	3,379	6,289	13,782	5,472	5,920	124,726	39,458

Tabulka 24: Čas zpracování dotazů nad tabulkou *Pocasi* s počtem záznamů 50 mil. (A), 75 mil. (B) a 100 mil. (C).

Z výsledků porovnání rychlostí zpracování dotazů (tabulky 23 a 24) nad tabulkou stejné velikosti lze vidět, že úměrně počtu záznamů roste i délka zpracování. Nejvyšší časy u testovaných dotazů se vyskytnou v případě využití agregační funkce *AVG*, tedy aritmetický průměr. Je to dáno tím, že tato agregační funkce na rozdíl od funkcí *SUM*, *MAX* či *MIN* nesplňuje podmínku asociativnosti a komutativnosti a není tedy provedena již v mapovací fázi, tzn. data nejsou výrazně zredukována. V tomto případě, kdy do fáze redukce přechází velké množství dat, je provedeno externí třídění, které pravděpodobně patří mezi úzká místa zpracování z důvodu postupného načítání dat z disku.

5.5 Testování vybrané open-source implementace MapReduce a porovnání

V této části testování jsou spuštěny výše uvedené dotazy nad nerelačním databázovým systémem *MongoDB*, verzí 2.6.0. Tento systém byl vybrán pro porovnání z důvodu uváděné jednoduchosti nasazení a především z toho důvodu, že patří mezi nejpoužívanější NoSQL databáze[9]. Stručný popis systému je uveden v podkapitole 3.3. Podle [26] lze očekávat nižší výkon zpracování v porovnání se systémem implementovaným v rámci této práce. Toto je způsobeno především použitím virtuálního stroje Javascriptu při zpracování map-reduce úloh a potřebnou deserializací dokůmentů z BSON formátu do JSON. Pro testy bylo nutno implementovat mapovací funkce a funkce redukce v javascriptu, které jsou ekvivalentní dotazům použitých v testování. Vzhledem k tomu, že práce není zaměřena na optimalizaci map-reduce úloh v MongoDB, je vhodné upozornit, že vlastní implementace mapovací funkce a funkce redukce nemusí být jediná správná možnost,

jak tyto funkce implementovat, a proto tato skutečnost může mít vliv na čas zpracování úlohy.

Testovací prostředí je totožné jako u předchozích testů, je tedy využito 10 uzlů. Rozdělení dat na uzly není řešeno replikací jako v implementovaném systému (plná replikace), ale pomocí tzv. *shardingu*. Sharding v MongoDB zajišťuje horizontální fragmentaci podle určitého vzoru. Ten byl u obou tabulek nastaven tak, aby podle atributu ID byly záznamy tabulek rovnoměrně rozděleny. Sharding v MongoDB a provedení MapReduce úlohy v distribuovaném prostředí vyžaduje značnou míru konfigurace. Na každém uzlu je nutné spustit *Shard Server*, který je zodpovědný za operace nad samotnými daty v uzlu. Dále je nutné spustit *Config Server* uchovávající metadata o rozložení dat na jednotlivých uzlech, a na závěr je potřeba spustit proces tzv. *Query Router*, na který jsou zasílány dotazy klienta. Query router se dotazuje Config serveru na rozmístění dat vzhledem k dotazu a následně se dotazuje na samotná data Shard serverů. Toto je rozdíl oproti vlastní implementaci modelu, kde žádný uzel nezná data jiného uzlu. Nicméně při spuštění map-reduce úlohy není tato znalost rozložení dat využita neboť úlohou musí projít všechna data.

Koncepce zpracování map-reduce úlohy v distribuovaném prostředí MongoDB je mírně odlišná než je v rámci této práce popsána a implementována [28]. Jednotlivé uzly nejsou rozdělovány na výše popisované mapovací uzly a uzly redukce, kde odděleně probíhají fáze zpracování. Úloha je zpracována tak, že mapovací funkce i funkce redukce je provedena na stejném uzlu. Tzn. že v našem testovacím prostředí se 10x provede kompletní map-reduce úloha, jejíž výsledky jsou sjednoceny Query Routerem. Toto je další rozdíl oproti původnímu modelu.

Oproti vlastní implementaci, kdy uzel obsahoval repliky všech dat a zpracovával pouze patřičné datové bloky, u MongoDB se zpracovávají veškerá data (fragment celkových dat) na uzlu. Ve výsledku je však objem zpracovaných dat konkrétním uzlem stejné. Pro porovnání byla použita verze tabulky *Zamestnanec* s 50 mil. záznamy a nad ní dříve testované dotazy, které zahrnovaly mapovací fázi i fázi redukce. Velikost kolekce v této databázi je 28,7 GB, což je v porovnání s importovaným csv souborem o velikosti necelých 10 GB značný nárůst a tato skutečnost také může negativně ovlivnit rychlost zpracování dotazu. Rychlost zpracování dotazů v MongoDB s porovnáním rychlosti zpracování vlastní implementace je uvedena v tabulce 25.

dotaz	**Q3**	**Q4**	**Q5**	**Q6**
počet výs.	1	1	51	51
	čas zpracování dotazu [s]:			
MongoDB	558	540	789	729
DP	6,416	10,189	178,820	54,260

Tabulka 25: Porovnání rychlosti zpracování vybraných dotazů v databázi MongoDB a vlastní implementací (DP) nad tabulkou *Zamestnanec* s počtem záznamů 50 mil.

Délka map-reduce zpracování úlohy v MongoDB je o výrazně delší v porovnání s vlastní implementací. Lze to přisuzovat především zmiňovanému použití Javascriptu pro zpracování úloh a také potřebnou deserializací BSON objektů. Při vhodné optimalizaci

implementovaných map-reduce úloh, případně i jiných konfiguračních hodnot databáze, jako je velikost cache, velikost rozdělení dat či další paralelizací každého uzlu, by se mohlo dosáhnout rychlejšího zpracování. Pro důkladné srovnání systémů by bylo vhodné využít větší množství tabulek s různým typem dat a pokud možno také využít všechny možnosti každého srovnávaného systému.

MongoDB databáze není založena čistě pro zpracování MapReduce úloh, je to jen nádstavba systému, kterou není vždy vhodné použít z důvodu rychlosti dotazu, neboť existuje např. integrace MongoDB s frameworkem Hadoop [26] nebo *Aggregation Pipelines* [27], což je engine pro spouštění agregovaných dotazů, který je implementován v C++ a pro jeho použití není nutná znalost JavaScriptu. Tyto příklady jsou časo uváděny jako vhodnější způsob zpracování agregovaných dotazů v MongoDB. Použití MapReduce v MongoDB se jeví efektivnější až při hlubší znalosti problematiky optimalizace map-reduce zpracování.

5.6 Shrnutí provedených testů

Po vyhodnocí každého provedeného experimentu byly identifikováno několik bodů charakterizující zpracování dotazů systémem s vlastní implementací modelu MapReduce.

1. Indexace vhodných atributů přinese mnohem efektivnější zpracování dotazů, obzvlášť u dat s počtem záznamů desítky milionů nebo vyšší. Je však podstatné analyzovat vhodnost každého vytvářeného indexu z důvodů následného většího objemu celkových zpracovávaných dat. Test č. 1 ukázal, že ne vždy může být vytvoření indexu přínosem.

2. Metadata popisující délků řetezců každého záznamu je vhodné využívat jen v případě vysoké propustnosti datové sítě a předpokládaným častým využitím jen vybraných dotazů. Velikost těchto metadat nevyváží dosažené zefektivnění zpracování. U některých typů dotazů (především dotazy s agregační funkcí) bylo zpracování urychleno o více než deset procent, je to však na úkor zmiňovanému většímu objemu dat a zhoršení efektivity jiných typů dotazů. je spíš vhodnější VSM data nepoužívat a místo toho využít vhodnou indexaci.

3. Vstupní data je vhodné rozdělovat spíš do větších bloků, tzn. 256 MB a víc. Při menším rozdělení se na času zpracování negativně podepisuje časté načítání dat z pevného disku. Zároveň nám ale větší datové bloky mohou přinést nevýhodu při nevyváženém hardwarovém výkonu uzlů. Master uzel zasílá novou mapovací úlohu vždy po přijetí oznámení ukončení předešlé mapovací úlohy z konkrétního uzlu. Čím výkonnější uzel je, tím více zpracuje datových bloků. Pokud by se tedy v clusteru nacházely např. dva uzly, z nichž jeden zpracuje za stejný čas čtyřikrát větší množství dat, je vhodnější vytvářet datové bloky měnší z důvodu rovnoměrného vyvážení zátěže uzlů.

4. Vždy je vhodné využít pro zpracování všechny uzly, které jsou k dispozici, tzn. hodnoty M a R nastavit na počet uzlů. Testy potvrdily předpoklad efektivnějším

zpracováním při využití větší paralelizace. Pouze v případě SQL dotazu, kdy není ve zpracování využita fáze redukce, má navyšování hodnoty R minimální vliv.

5. Vyšší paralelizace zpracování lze u některých typů dotazů, především u dotazů bez agregační funkce, dosáhnout podobných časů při několikanásobně větších objemech dat. V případě agregačních funkcí se dotazy stávají náročnějšími především díky seskupování záznamů dle zvolených atributů.

Díky těmto základním testům si lze vytvořit základní pohled na implementovaný systém. Rychlost zpracování je v porovnání s MongoDB lepší a pro případné další rozšiřování systému by bylo vhodné provést rozsáhlejší testy nad optimalizovaným systémem. Po detailnějším studiu problematiky MongoDB MapReduce se tato databáze nyní nejeví jako vhodná k porovnání s vlastní implementací, protože pro dosažení rychlejšího zpracování vyžaduje poměrně složitou optimalizaci, navíc koncept zpracování map-reduce úloh je mírně odlišný od popisovaného modelu. Proto by bylo vhodnější porovnat systém, který je založen čistě na MapReduce, jako implementace v této práci, a nabízí podobný dotazovací jazyk jako SQL. Mohlo by se jednat např. o systém Hive [12], podporující dotazovací jazyk HiveQL.

6 Závěr

V úvodu práce byla teoreticky probrána problematika zpracovaní dat modelem MapReduce. Dále byly popsány vybrané databázové systémy s vlastní implementací modelu. Díky těmto částem byly identifikovány klíčové vlastnosti a jednotlivé výhody či nevýhody vyplývající z modelu nebo různorodosti vybraných systémů. Tyto poznatky pak posloužili jako východiska při návrhu vlastního systému implementující model MapReduce. Implementovaný systém svou funkcionalitou nepokrývá rozsáhlost popisovaných či osatních systémů, nicméně obsahuje základní funkcionalitu, která je postavena na popisovaném modelu, a je tak umožněno provádět testování a porovnání s ostatními implementacemi modelu.

Vývoj aplikace probíhal iterativně, kdy postupně byl systém rozšiřován o nové možnosti, a zároveň probíhala i částečná reimplementace zpracování neboť se ukázalo, že určité stávající řešení nebylo vhodné pro řešení některých problémů související ať už s modelem MapReduce nebo obecně s distribuovaným zpracováním dat či se správou paměti. Jedním z problémů bylo řešení fyzického uložení dat, kdy v původním návrhu bylo použití nepředzpracovaných csv souborů, případně jen částečně pro indexaci. Toto řešení bylo velmi neefektivní a bylo nutné přistoupit k vlastní fyzické implementaci uložení dat pro dosažení větší efektivity a rozumných času při zpracování objených dat. Jedním z dalších podstatných problémů řešených při implementace byla správa paměti. Při zpracování dat o velikostech alespoň jednotky GB vyskytl problém s fragmentací paměti, tzn. situace kdy v paměti neexistoval souvislý blok požadované velikosti. Tato chyba jasně ukazovala na nevhodnou správu, respektive častou alokaci nové paměti, a proto bylo nutné maximalizovat využití již alokované paměti.

V závěrečné části proběhlo otestování implementace nad vybranými daty, které potvrdily vhodnost návrhu (např. zavedení indexace) nebo naopak ukázaly na nevhodný návrh jiné části (např. VSM metadata). Testy zároveň ukázaly, že i implementací modelu MapReduce v rámci diplomové práce, kdy bylo nutné některé části vynechat nebo se jimi nezabývat příliš detailně, lze dosáhnout rozumných časů zpracování dat o velikosti až 20 GB.

V porovnání se systémem MongoDB dopadla vlastní implementace výrazně lépe, obzvlášť u dotazů bez provádění seskupení, respektive agregace v rámci celé tabulky. MongoDB provádí map-reduce zpracování velmi pomalu, což může být dáno použitím JavaScriptového enginu, nutností deserializací formátu BSON v průběhu zpracování, nebo také absence nutné optimalizace či vhodné konfigurace, která v rámci porovnání nebyla provedena. Jak bylo zmíněno dříve, další srovnání by bylo vhodné provádět se systémy využívají pro map-reduce zpracování vlastní nebo převzatý dotazovací jazyk. Odpadla by tak velká závislost rychlosti zpracování na způsobu implementace požadované mapovací funkce a funkce redukce.

Během vývoje bylo získáno obrovské množství nových znalostí ohledně probírané problematiky, i znalostí ohledně samotného programování v jazyce C++, a proto při zpětném pohledu na návrh aplikace by bylo vhodné uvést některá doporučení na změny či rozšíření funkcionality, které se nyní jeví jako vhodnější než obsahuje původní návrh.

Může se jednat např. o:

- Rozšíření o další funkcionality SQL, kupříkladu o použití skalárních funkcí, klauzule HAVING či ORDER BY, případně o spojování tabulek.
- Změna návrhu reprezentace SQL dotazu. Stávají návrh svým rozsahem postačuje pro implementovanou funkcionalitu, nicméně při rozšiřování by bylo potřeba upravit návrh reprezentace.
- Použití paměťového poolu s vlastní správou. Zajímavým řešením by byla alokace většího bloku paměti, která by se přidělovala nově vytvářeným strukturám podle potřeby. Toto řešení by minimalizovalo problém, který nastává u časté alokace, a sice nemožnosti alokovat požadovanou velikost z důvodu vyšší fragmentace paměti.
- Efektivnější implementace fyzického uložení dat. Jak bylo dříve zmíněno, vhodnější fyzická implementace dat měla výrazný vliv na efektivitu zpracování. I přes svou jednoduchost bylo dosaženo rychlejších času při popisovaném uložení dat. Při sofistikovanější implementaci fyzického uložení např. do B-stromu by bylo možné dosáhnout zas o něco efektivnějšího zpracování. To stejné platí pro indexy.
- Vytvoření grafického uživatelského prostředí. Vzhledem k náročnosti implementace byla vytvořena pouze jako konzolová aplikace. Pro širší a přehlednější využití je vhodné implementovat grafické rozhraní aplikace ať už ve formě desktopové nebo webové aplikace.

Určitě by se našly další úpravy, které by přinesly lepší přehlednost či efektivitu implementovaného systému, nicméně i stávající implementace může posloužit k základnímu dotazování dat, a zároveň je vhodným zdrojem informací pro další studia problematiky distribuovaného zpracování objemných dat nejen modelem MapReduce.

Adam Babušek

7 Reference

[1] Dean Jeffrey, Ghemawat Sanjay, *MapReduce: Simplified Data Processing on Large Clusters*, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, USA, 2004.

[2] Pavlo Andrew, Paulson Erik, Rasin Alexander, Abadi Daniel J., DeWitt David J., Madden Samuel, Stonebraker Michael. *A Comparison of Approaches to Large-Scale Data Analysis*, Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, New York, 2009.

[3] Tom White *Hadoop: The Definitive Guide*, Sebastopol, CA: O'Reilly, 2009, 501 s., ISBN 05-965-2197-9.

[4] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, HPCA '07 Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Washington, 2007.

[5] Sherif Sakr, Anna Liu, Ayman G. Fayoumi *The Family of MapReduce and Large Scale Data Processing Systems*, ACM Computing Surveys (CSUR), 2013.

[6] webové stránky organizace Apache Software Foundation *Apache Software Foundation*, http://www.apache.org

[7] webové stránky projektu Hadoop MapReduce *Hadoop MapReduce*, http://hadoop.apache.org/mapreduce

[8] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, Utkarsh Srivastava, *Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*, VLDB 2009, Lyon, France.

[9] webové stránky projektu databáze MongoDB *MongoDB*, http://mongodb.org

[10] webové stránky databáze CouchDB *CouchDB*, http://couchdb.apache.org

[11] webové stránky programovacího jazyku Erlang *Erlang*, http://erlang.org

[12] Thusoo Ashish, Sarma Joydeep Sen, Jain Namit, Shao Zheng, Chakka Prasad, Anthony Suresh, Liu Hao, Wyckoff Pete, Murthy Raghotham, *Hive - A Warehousing Solution Over a Map-Reduce Framework*, Proceedings of the VLDB Endowment 2009, Lyon, France.

[13] George Lars, *HBase: The Definitive Guide*, CA: O'Reilly, 2011, 556 s. ISBN 14-493-9610.

[14] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, Eugene J. Shekita, *Jaql: A Scripting Language for*

Large Scale Semistructured Data Analysis, Proceedings of the VLDB Endowment 2011, Seattle, USA.

[15] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, *Interpreting the Data: Parallel Analysis with Sawzall*, Scientific Programming - Dynamic Grids and Worldwide Computing, 2005.

[16] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, Michael Wong, *Tenzing: A SQL Implementation On The MapReduce Framework*, Proceedings of the VLDB Endowment 2011, Seattle, USA.

[17] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, Xiaodong Zhang, *YSmart: Yet Another SQL-to-MapReduce Translator*, Department of Computer Science and Engineering, The Ohio State University.

[18] Nedbálek Aleš, *Distribuované datové struktury pro masivně paralelní zpracování dat*, diplomová práce, Fakulta elektrotechniky a informatiky, Katedra informatiky, VŠB-TUO, r. 2013.

[19] Diana Lorentz, Mary Beth Roeser, *Oracle Database SQL Language Reference, 11g Release 2*, E41084-02, 2013.

[20] Krátký Michal, Bača Radim, *Databázové systémy* [pdf dokument], dostupné z *http://dbedu.cs.vsb.cz/SubPages/OpenFile/OpenFile.aspx?file=dbcb/dbcb.pdf* Fakulta elektrotechniky a informatiky, katedra informatiky, VŠB - TUO, Ostrava, 28.9. 2010.

[21] Liu Ling, M. Ozsu, *Encyclopedia of database systems*, New York: Springer, 2009, 53749 s. ISBN 978-038-7496-160.

[22] Prata Stephen, *Mistrovství v C++, 2. aktualizované vydání*, Computer Press, a.s., 2004, 1006 s., ISBN 802-5100987.

[23] David S. Touretzky, *COMMON LISP: A Gentle Introduction to Symbolic Computation*, Dover Publications, aktualizované vydání, 2013, 600s., ISBN 978-0486498201.

[24] Foto N. Afrati, Jeffrey D. Ullman, *Optimizing Joins in a Map-Reduce Environment*, Proceedings of the EDBT 2010, Lausanne, Switzerland.

[25] Nikos Ntarmos, Ioannis Patlakas, Peter Triantafillou, *Rank Join Queries in NoSQL Databases*, Proceedings of the VLDB Endowment 2014, Hangzhou, China.

[26] Elif Dede, Madhusudhan Govindaraju, Dan Gunter, Shane Canon, Lavanya Ramakrishnan *Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis*, In proceeding of: 4th Workshop on Scientific Cloud Computing (ScienceCloud), 2013, New York, USA.

[27] MongoDB Documentation Project, *MongoDB Aggregation and Data Processing*, release 2.6.0, duben 2014.

[28] B. Rama Mohan Rao, A. Govardhan *Sharded Parallel Mapreduce in Mongodb for Online Aggregation*, International Journal of Engineering and Innovative Technology (IJEIT), 2013.

[29] dokumentace programovacího jazyka Java verze 8 na stránkách společnosti Oracle *Java 8*, http://docs.oracle.com/javase/8/docs/

[30] Vatika Sharma, Meenu Dave, *SQL and NoSQL Databases*, International Journal of Advanced Research in Computer Science and Software Engineering, 2012.

[31] webové stránky specifikace formátu *JSON*, http://www.json.org

[32] Paul C. Zikopoulos, Chris Eaton, Dirk deRoos, Thomas Deutsh, George Lapis *Understanding Big Data: Analytict for Enterprise Class Hadoop and Streaming Data*, McGraw-Hill Osborne Media, 2011, 166 s., ISBN 9780071790536.

[33] webové stránky Python MapReduce framworku Disco *Disco*, http://www.discoproject.org/

[34] webové stránky Ruby MapReduce framworku Skynet *Skynet*, http://www.skynet.rubyforge.org/

[35] Justin Talbot, Richard M. Yoo, Christos Kozyrakis *Phoenix++: Modular MapReduce for Shared-Memory Systems*, MapReduce '11 Proceedings of the second international workshop on MapReduce and its applications 2011, New York, USA.

[36] webové stránky společnosti EMC2 a databáze GreenPlum Database *GreenPlum Database*, http://www.emc.com/products/family/greenplum-database.htm

[37] webové stránky Teradata a databáze Teradata Aster MapReduce *Teradata Aster MapReduce*, http://www.teradata.com/Aster-MapReduce-Appliance/

[38] webové stránky společnosti Amazon a webové služby Amazon Elastic MapReduce *Amazon Elastic MapReduce*, https://aws.amazon.com/elasticmapreduce/

[39] webové stránky generátoru dat Spawner Data Generator *Spawner Data Generator*, http://spawner.sourceforge.net/

A Konfigurace a spuštění implementované aplikace

Výběr režimu

Aplikace nevyžaduje instalaci, postačí vytvoření nového adresáře, do kterého se umístí spustitelný soubor aplikace. Případné další složky si aplikace vytvoří sama.

Aplikaci lze spustit ve dvou režimech – klient nebo server. Po spuštění aplikace je uživateli nabídnuta volba režimu. V případě výběru klienta použijeme příkaz *client*. V případě volby serveru použijeme příkaz *server <port>*. Volba portu je pro nastavení portu, na kterém bude prováděno naslouchání pro TCP protokol. Volba to není povinná, defaultní port je 40001.

Po spuštění aplikace v server režimu je nutné zadat příkaz *start*, který spustí naslouchání serveru. Změnit port lze příkazem *port <cislo_portu>*. Předtím však musí být server zastaven příkazem *stop*. Příkaz *exit* provede ukončení celé aplikace.

Konfigurace serveru

Při prvním spuštění je nutné zadat příkaz *server_set*. Tento příkaz konfiguruje server v rámci použitého frameworku. V postupných krocích je nutné zadat název serveru, vybrat jednu z dostupných IPv4 adres pro identifikaci uzlu, broadcastovou adresu a id. Po provedení nastavení je nutné server restartovat. Toto nastavení je nutné provést vždy, kdy dojde ke změně IP adresy.

List dostupných serverů

Aby klient věděl, na který server má zasílat dotazy, musí být server přidán do souboru *C_SERVERLIST.txt*, což je csv soubor, ve kterém se uvádí u server vždy id, název, IPv4 adresa a port, na kterém server naslouchá. Serverů může být v případě klienta zapsaných více. Aplikace se pokouší postupně připojit k následujícím serverům, pokud předchozí nejsou dostupné. Klientem dotázaný server se v MapReduce zpracování stává masterem uzlem.

Stejný seznam, tentokrát v souboru *S_SERVERLIST.txt*, musí být vytvořen na každém serveru, na který se klient připojuje. Tento seznam pak představuje seznam pracovních uzlů, mezi které master uzel rozděluje úkolu při MapReduce zpracování. Pracovní uzly tento seznam mít nemusí. Pokud je však požadavkem lepší dostupnost systému, je vhodné tento seznam uvádět i na pracovních uzlech, aby se tak kdykoliv mohli stát master uzlu v případě nedostupnosti jiného. Samozřejmě pak musí být uvedeny v seznamu serverů klienta.

Pro promítnutí změn v seznamu je nutné server nebo klienta restartovat.

Import dat

Pro vytvoření dat v systému je nutné provést import z textové souboru v CSV formátu. Hodnoty musí být odděleny znakem ';' a pro správný import by měl být soubor v kódování UTF-8. K datovému souboru musí být dodán další textový soubor s jednoduchým

popisem schématu – na jednom řádku se vždy nachází název atributu a jeho datový typ (v systému jsou implementovány čtyři datové typy - INTEGER, DOUBLE, VARCHAR, DATETIME) oddělný dvojtečkou, např:

```
id:int
jmeno:varchar
vaha:double
cas:datetime
```

Pak je následně možný import pomocí příkazu *import zamestnanec.csv schema.txt 256*.Vytvořená tabulka má pak v systému název stejný jako jméno souboru bez přípony. Poslední hodnota značí velikost datového bloku v MB, do kterých jsou data rozdělována. Uvedení této hodnoty není povinné, defaultně je nastavena na 128 MB. Tento import dat je nutno provést na každém uzlu, pokud bude využito distribuované zpracování, případně lze provést manuální zkopírování obsahu vytvořeného adresáře *data*.

Vytvoření indexů

Systém umožňuje pro rychlejší zpracování vytvářet indexy nad vybranými aributy. Indexy lze nad tabulkou vytvořit po jednom zadáním názvu tabulky a pořadí atributu v tabulce (kdy pořadí prvního atributu je 0). Příkaz pak může vypadat takto: *create_index zamestnanec 3*.

Distribuované prostředí

Pro spuštění dotazu je nutné spustit minimálně tři instance aplikace. První je klient, druhá je master uzel a třetí, případně další, jsou pracovní uzly. Samotné zpracování pak proběhne na pracovním uzlu. Při spouštění instancí serverů je správně nastavit použitý port, musí odpovídat nastaveným portům v listu serverů klienta nebo master uzlu.

Spuštění dotazu

Před spuštěním dotazu jsou použity tři příkazy:

- *m <hodnota_m>* - nastaví hodnotu M, tedy počet vytvořených mapovacích uzlů. Hodnota je povinná, minimální hodnota je 1, maximální hodnota je počet datových bloků dotazované tabulky.
- *r <hodnota_r>* - nastaví hodnotu R, tedy počet vytvořených uzlů redukce. Hodnota je povinná, minimální hodnota je 1, maximální hodnota je počet pracovních uzlů v distribuovaném prostředí.
- *sql <sql_dotaz>* - nastaví samotný SQL dotaz zaslaný serveru.

Nastavení těchto hodnot lze vypsat příkazem *settings*. Jakmile jsou hodnoty nastaveny, lze provést příkaz *start*, kdy se vyhledá ze seznamu serverů první dostupný a zašle se na něj dotaz. Jakmile je dotaz zpracován, je uživatel informován zprávou s celkovým počtem výsledků. Příkazem *result* pak proběhne jejich výpis.

B Obsah přiloženého CD

- **zdrojové kódy** - složka */src* obsahuje veškeré zdrojové kódy (vlastní implementace + použitý framework) potřebné k projektu do MS Visual Studio. Zdrojové kódy vlastní implementace se nachází ve složce */src/test/mbt/bab168* ve formě zmiňovaného projektu.
- **aplikace** - složka */bin* obsahuje spustitelný soubor implementované aplikace
- **text** - složka */text* obsahuje dokument pdf této práce